Systems
Optimization
Laboratory

Primal Barrier Methods For Linear Programming

by
Aeneas Marxen

TECHNICAL REPORT SOL 89-6

June 1989

Department of Operations Research
Stanford University
Stanford, CA 94305

4

SYSTEMS OPTIMIZATION LABORATORY
DEPARTMENT OF OPERATIONS RESEARCH
STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305-4022

Primal Barrier Methods For Linear Programming

by
Aeneas Marxen

TECHNICAL REPORT SOL 89-6

June 1989

Research and reproduction of this report were partially supported by the U.S. Department of Energy Grant DE-FG03-87ER25030, and Office of Naval Research Contract N00014-87-K-0142.

PRIMAL BARRIER METHODS FOR LINEAR PROGRAMMING

Aeneas Marxen, Ph.D.
Stanford University, 1989

The linear program $\min c^T x$ subject to $Ax = b,\ x \geq 0$, is solved by the *projected Newton barrier method*. The method consists of solving a sequence of subproblems of the form $\min c^T x - \mu \sum \ln x_j$ subject to $Ax = b$. Extensions for upper bounds, free and fixed variables are given. A linear modification is made to the logarithmic barrier function, which results in the solution being bounded in all cases. It also facilitates the provision of a good starting point. The solution of each subproblem involves repeatedly computing a search direction and taking a step along this direction. Ways to find an initial feasible solution, step sizes and convergence criteria are discussed.

Like other *interior-point method* for linear programming, this method solves a system of the form $AH^{-1}A^T q = y$, where H is diagonal. This system can be very ill-conditioned and special precautions must be taken for the Cholesky factorization. The matrix A is assumed to be large and sparse. Data structures and algorithms for the sparse factorization are explained. In particular, the consequences of relatively dense columns in A are investigated and a Schur-complement method is introduced to maintain the speed of the method in these cases.

An implementation of the method was developed as part of the research. Results of extensive testing with medium to large problems are presented and the testing methodologies used are discussed.

Gewidmet meinen Eltern,

Gisela und Aeneas Marxen,

deren Liebe, Großzügigkeit und Ermutigung

ich alles zu verdanken habe.

Introduction

From the beginning, linear programming problems have played a central role in Operations Research. Discovered by George B. Dantzig in 1947, the simplex method in its many variations has evolved as the standard algorithm for linear programming. For a linear program (LP) that has a solution, there usually exists an optimal point at a vertex of the feasible region. The iterates of the simplex method move along the boundary of the feasible region to find such a vertex. The simplex method can be shown to require a non-polynomial number of iterations for a contrived class of problems, although in practice it tends to need a number of iterations that is little more than linear in the problem dimensions.

A number of alternatives to the simplex method that generate iterates in the interior of the feasible region, were proposed early on. Among them was the barrier method. (For a complete discussion of barrier methods, see Fiacco [Fia79]. Classical barrier and penalty methods are described in Fiacco and McCormick [FM68]. Fletcher [Fle81] and Gill, Murray and Wright [GMW81] give overviews of barrier and penalty methods.) The logarithmic barrier function considered in this thesis was first suggested by Frisch [Fri54,57]. It was utilized to devise a sequence of nonlinear, unconstrained subproblems for solving linear programs by Parisot [Par61]. Osborne [Osb72] and Wright [Wri76] added an active set strategy to the method, an idea not followed in this research. Gill *et al.* [GMSTW86] proposed using Newton's method to solve individual subproblems.

Although the number of subproblems has been observed to be small, the nonlinearities involved make them hard problems to solve. Additionally there is a certain *minimum* number of subproblems, irrespective of problem size. Since at the outset the only problems solved were small by today's standards, the barrier method was not considered to be competitive with the simplex method at that time. Interest revived recently, however, when improvements in design and performance of computers and improved algorithms for factorizing sparse matrices made interior-point methods an alternative worthy of serious consideration.

The spark for this renewed interest came when Karmarkar [Kar84] demonstrated that the combinatorial complexity of finding the optimal vertex can be overcome by solving a

series of nonlinear optimization problems whose optimal point is interior. Subsequently it was shown that the algorithm he used is closely related to one proposed by Dikin [Dik67]. The theoretical question, whether a linear programming algorithm with only polynomial complexity can be found, had been resolved earlier when Khachiyan [Kha79] analysed his method based on an algorithm of shrinking ellipsoids [Shor77].

It is now generally recognized that essentially all interior-point methods for linear programming inspired by Karmarkar's projective method are closely related to application of Newton's method to a sequence of barrier functions (see [GMSTW86j). Newton's method is based on minimizing a local quadratic model of the barrier function derived from first and second derivative information at the current iterate. Unfortunately, several difficulties can arise because of the nature of barrier functions. The extreme nonlinearity of the barrier term near the boundary means that a quadratic model may be accurate only in a very small neighborhood of the current point. For a degenerate linear program, the system of equations that has to be solved becomes increasingly ill-conditioned. Finally, the strictly interior starting point that this method requires, may be inconvenient or impossible to obtain.

Recent publications (e.g. [ARV86], [MM87], [VMF86]) compare implementations of interior-point methods to one of the simplex method and show impressive reductions in computing time for a certain set of problems. However, there has been little interest in comparing different interior-point methods, and hardly any evidence is given concerning their reliability. While interior-point methods seem similar enough that their comparison can be safely left for future research, the issue of reliability is an important one. The question of whether interior-point methods are fit to serve as an all-purpose replacement of the simplex method for general linear programs, remains unanswered.

The intention of the research presented in this dissertation is to explore the behavior of the barrier method when solving real-world, medium-to-large problems and to develop ways of overcoming the obstacles encountered. As a general guideline, we have attempted to develop the fastest algorithm that would be able to deal with the numerical difficulties arising from degeneracy, rank-deficiency and other characteristics that make real-world problems hard to solve. More importantly, we have tried to identify those areas where a trade-off between speed and reliability must be made. The test set consists of the first 53 problems of the *netlib* collection [Gay85], which was formed as a benchmark for comparing linear programming algorithms. At the outset of this research, no complete set of results for these problems had been published for the new class of interior-point methods.

To make comparisons with the simplex method as meaningful as possible, an implementation was developed that operates under the same conditions as the simplex code to which

it was compared. In particular, both implementations work with the same constraint matrix, require about the same amount of memory and were produced using the same portable, high-level computer language. No assessment is made of whether enhancements in any of these three directions might benefit one method more than the other.

I have been privileged in that I was able to conduct this research in close collaboration with the SOL Algorithms Group in the Operations Research Department at Stanford. The discussions in the group and the extensive support I received from the associated researchers and students were very helpful. I would like to thank Prof. George B. Dantzig for serving on my doctoral committee, for two most interesting research seminars, and for giving me a perspective on the evolution of the field. Margaret H. Wright became important for this thesis almost unintentionally; she gave a lively and fascinating presentation on barrier methods for LP as part of the OR Colloquium series, and she provided an office with a computer workstation by being on leave throughout 1988. I am indebted to Prof. Michael A. Saunders for sharing his experience and answering many questions, often late at night, as well as for providing the `MINOS` subroutines. My thesis advisor, Prof. Walter Murray, will be fondly remembered for his many invaluable suggestions, his humor and his generosity with signatures for all my forms. And last, but not least, I would like to thank Prof. Philip E. Gill for his time, patience and availability when helping me with my questions. References to "P. E. Gill (1987, 1988). *Private communication.*" were omitted from this manuscript, since they might have rendered certain parts all but unreadable. If this dissertation turns out to be readable and helpful, it is largely owing to his proofreading, whereas the idiosyncracies and shortcomings are solely mine.

A. Mx.

Part I The Algorithm

Chapter 1

What is the problem ?

The linear program considered is of the following standard form:

$$\begin{array}{lll} \text{SLP} & \underset{x}{\text{minimize}} & c^T x \\ & \text{subject to} & Ax = b \\ & & x \geq 0. \end{array}$$

The vector $x \in \Re^n$ contains the decision variables, $c \in \Re^n$ contains the weights of the objective function. The matrix $A \in \Re^{m \times n}$ is called the constraint matrix and is assumed to be of full row rank. The vector $b \in \Re^m$ is called the right-hand side. The feasible region of the problem is assumed to have a nonempty interior, so that there exists an x such that $Ax = b$ and $x > 0$.

The constraint matrices of the problems to be considered are large (up to 10000 columns) and very sparse (90%–99% of the elements are zero).

We want to find a solution x^* of this problem by solving a sequence $k = 1, 2, \ldots$ of *barrier-function subproblems.* Here, the nonnegativity constraints are no longer stated explicitly, but are enforced implicitly by the objective function. A barrier subproblem is of the form

$$\begin{array}{ll} \underset{x}{\text{minimize}} & F^k(x) = c^T x + \sum_j f_j^k(x_j) \\ \text{subject to} & Ax = b. \end{array}$$

With the proper choice of $F^k(x)$, the sequence of solutions $x^*(k)$ of these subproblems converges to the solution x^* of the original problem.

Since a second-derivative method is used to solve each subproblem, we shall define $g(x) = \nabla F(x)$ and $H(x) = \nabla^2 F(x)$ to be the gradient and the Hessian of the objective

function. (The subproblem index k is omitted for clarity, unless needed.) We denote

$$g = c + g_B, \quad g_{Bj} = \partial f_j / \partial x_j$$

and

$$H = \operatorname{diag} h, \quad h_j = \partial^2 f_j / \partial x_j^2, \quad j = 1, \ldots, n.$$

The functions f_j are defined to be strictly convex over the interior of the feasible region, so that $h_j > 0$ for all j and H^{-1} exists. Note that $F(x)$ is separable so that the Hessian H is a diagonal matrix and its inverse H^{-1} is readily computable as

$$H^{-1} = \operatorname{diag}(1/h_1, \ldots, 1/h_n).$$

The Lagrangian function associated with the subproblem is $F(x) - \pi_b^T(Ax - b)$, where π_b denotes the Lagrange multipliers of the constraints $Ax = b$. The first-order necessary condition for optimality is that the gradient of the Lagrangian at $x^*(k)$ must vanish, i.e.,

$$g - A^T \pi_b = 0.$$

The Projected Newton Method

To solve the subproblem, a *feasible-point descent method* is employed. Every iterate x satisfies the constraints $Ax = b$, and the next point x' is found on a search direction p, so that $x' = x + \alpha p$. Convergence is ensured by choosing p as a descent direction, and α such that the objective function value $F(x')$ is sufficiently smaller than $F(x)$ (see page 12). Feasibility is ensured by satisfying $Ax^0 = b$ for the initial point and the null-space condition $Ap = 0$.

The *Newton search direction* satisfies these conditions and is computed using second-derivative information. The direction is defined as the step to the minimizer of a quadratic approximation to $F(x)$ on the feasible region, as derived from the local Taylor series. Thus p is the solution of the quadratic program

$$\begin{aligned} \underset{p}{\text{minimize}} \quad & g^T p + \tfrac{1}{2} p^T H p \\ \text{subject to} \quad & Ap = 0. \end{aligned}$$

The vector p satisfies the QP-optimality condition

$$g + Hp - A^T \pi = 0,$$

where π is the vector of Lagrange multipliers associated with the equality constraints $Ap = 0$. Since $p \to 0$ as $x \to x^*(k)$, the Lagrange multipliers π converge to the multipliers π_k of the original problem.

Note that $p = 0$ is feasible for the QP, so that the optimal objective function value is not positive and $g^T p \leq -\frac{1}{2} p^T H p < 0$ for the optimal p.

The null-space condition and the QP-optimality condition can be summarized in the *Karush-Kuhn-Tucker* (KKT) system,

$$\begin{pmatrix} H & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} -p \\ \pi \end{pmatrix} = \begin{pmatrix} g \\ 0 \end{pmatrix}.$$

In our implementation, the KKT-system is solved by computing π from the positive-definite system

$$AH^{-1}A^T\pi = AH^{-1}g,$$

and by recovering the search direction as $p = H^{-1}(g - A^T\pi)$.

These equations are called *normal equations*, a name taken from a weighted least-squares problem that is equivalent to the KKT-system. Let D be a diagonal matrix such that $D^2 = H^{-1}$ and define a vector $r = -D^{-1}p$. Now r and π satisfy

$$\begin{pmatrix} I & DA^T \\ AD & 0 \end{pmatrix} \begin{pmatrix} r \\ \pi \end{pmatrix} = \begin{pmatrix} Dg \\ 0 \end{pmatrix},$$

so that π is the solution, and r the optimal residual, of

$$\underset{\pi}{\text{minimize}} \, \|D(g - A^T\pi)\|_2^2.$$

The derivative of this norm with respect to π is $2AD^2A^T\pi - 2AD^2g$. Solving for the zero of this derivative gives the normal equations.

The solution of the KKT-system is by far the most difficult aspect of using an interior-point method, both in terms of computational effort and in terms of numerical problems that must be dealt with. Exploiting sparsity in A is essential for the efficiency of the whole algorithm, and finding a way to deal with ill-conditioning in A and H is crucial for reliability. Part II will be devoted to these difficulties. For the rest of Part I we assume that a search direction p can always be computed.

The Newton step from x to $x' = x + \alpha p$ is sometimes referred to as a *minor iteration*. This is to distinguish it from a *major iteration*, which is the solution of one barrier subproblem. Unless stated otherwise, we will use the term *iterations* to refer to minor iterations.

The Logarithmic Barrier Function

A straightforward example of an objective function $F^k(x)$ is the logarithmic barrier function. The sequence of subproblems with decreasing barrier parameters μ is defined as

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & F^k(x) = c^T x - \mu^k \sum_j \ln x_j \\ \text{subject to} \quad & Ax = b. \end{aligned}$$

The first two derivatives of the logarithmic barrier function $F(x)$ are given by

$$g = c + g_B, \quad g_{Bj} = -\mu / x_j$$

and

$$H = \operatorname{diag} h, \quad h_j = \mu / x_j^2, \quad j = 1, \ldots, n.$$

The solutions $x^*(k) \equiv x^*(\mu^k)$ of the subproblems converge to x^* as $\mu = \mu^k \to 0$. To see that, multiply the optimality condition $g - A^T \pi_b(\mu) = 0$ with $x^*(\mu)$ to get

$$c^T x^*(\mu) + {g_B}^T x^*(\mu) - x^*(\mu)^T A^T \pi_b(\mu) = 0.$$

By the definition of g_B and the feasiblity of $x^*(\mu)$ this reduces to $c^T x^*(\mu) - b^T \pi_b(\mu) = \mu$. The multipliers $\pi_b(\mu)$ are feasible for the dual of the linear program (see [Dan63] for duality theory). Taking limits for $\mu \to 0$ shows that $c^T x^* - b^T \pi^* = 0$ which implies that x^* is optimal for the LP.

More precisely, it can be shown (see [Jit78],[JO78]) that

$$\|x^*(\mu) - x^*\| = O(\mu)$$

for primal nondegenerate systems and sufficiently small μ, and

$$\|x^*(\mu) - x^*\| = O(\sqrt{\mu})$$

for primal degenerate systems.

The function $x^*(\mu)$ is called the *barrier trajectory*. (See page 29 for strategies to choose barrier parameters μ^k.)

Equivalence relations between Karmarkar's projective method and the logarithmic barrier method using the projected Newton method have been established ([GMSTW86]) for a certain sequence of barrier parameters. A proof of polynomial complexity exists for the barrier method under certain (but different) conditions on the barrier parameter (see Gonzaga

[Gon87]). Renegar and Shub [RS88] show that an $O(\sqrt{m}L)$ bound holds for the number of iterations, which gives an $O(n^2m^{1.5}L)$ bound on the number of operations for the normal barrier method and $O(n^2mL)$ for a modified version. (The scalar L is used to denote the number of bits required to specify the problem.) This iteration bound is achieved, under some conditions on the starting point, by doing only one Newton iteration per subproblem and by updating μ according to $\mu^k = (1 - 1/(41\sqrt{m}))\mu^{k-1}$. Although the theoretical importance of these results is not doubted, it should be acknowledged that they provide little guidance for a practical implementation of the method. [1]

All barrier functions used in this research are close variations of the logarithmic barrier function as defined here. (For extensions see Chapter 2 and pages 24 and 34.)

Overview of the Algorithm

The main steps of the algorithm take the following form:

$x \leftarrow$ strictly feasible x^0, $\mu \leftarrow \mu^1$ and compute $g(x)$, $H(x)$;
repeat { Subproblem - major iteration }
  repeat { Newton step - minor iteration }
    Compute $r \leftarrow \sqrt{H^{-1}}(g - A^T\pi)$;
    Set *rg_conv* $\leftarrow \|r\|$ *sufficiently small*;
    if not *rg_conv* **then**
      Solve the KKT system for search direction p and multipliers π,

$$\begin{pmatrix} H & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} -p \\ \pi \end{pmatrix} = \begin{pmatrix} g \\ 0 \end{pmatrix};$$

      Find maximum step $\alpha_M = \max\{\alpha \geq 0 \mid x + \alpha p \text{ is feasible}\}$;
      Choose a steplength $\alpha \in (0, \alpha_M)$ that
        decreases the barrier function sufficiently;
      Update $x \leftarrow x + \alpha p$ and compute $g(x)$, $H(x)$;
    end;
  until *rg_conv*;
  Decrease μ;
until $\mu = \mu_{\min}$;

[1] In a crude test the smallest test problem **AFIRO** needed about 1100 iterations to converge when using this strategy, compared to 17 iterations when using a more practical alternative (see page 29).

The importance of the feasible starting point x^0 and its derivation are discussed in Chapter 3.

The logical variable *rg_conv* indicates the convergence of the reduced gradient $r = \sqrt{H^{-1}}(g - A^T\pi)$. For more discussion on the issue of convergence, both for the subproblems and for the whole problem, and on the way the μ^k are chosen, see Chapter 4.

The choice of α is described in the following section.

The Steplength

For a given search direction p, the objective function reduces to a univariate function $f(\alpha) = F(x + \alpha p)$. The distance to the closest bound along p is α_M, which implies that $f(\alpha_M) = \infty$ since the barrier function has a singularity at the boundary. The derivatives at the endpoints of the feasible interval $[0, \alpha_M]$ are $f'(0) = g^T p < 0$ and $f'(\alpha_M) = \infty$. Since $f(\alpha)$ is convex, there exists a unique $\alpha^* \in (0, \alpha_M)$ with $f'(\alpha^*) = 0$.

The computation of the steplength involves an iterative procedure for finding an α close to the zero of f'. Many efficient algorithms have been developed for finding the zero of a general univariate function (see, e.g. [Brent73]), based on iterative approximation by a low-order polynomial. However, such methods tend to perform poorly in the presence of singularities. In order to overcome this difficulty, special steplength algorithms have been devised for the logarithmic barrier function (e.g. [FM69], [MW76]). These special procedures are based on approximating $f'(\alpha)$ by a function with a similar type of singularity.

At each iteration an estimate α_j and an interval $I_j = [\underline{\alpha}_j, \overline{\alpha}_j]$ are generated, so that $\underline{\alpha}_j$ is the largest α encountered so far with $f'(\alpha) < 0$ and $\overline{\alpha}_j$ is the smallest α with $f'(\alpha) > 0$. The interval is initialized to $I_0 = [0, \alpha_M]$. The approximating function is of the form

$$\phi(\alpha) = \gamma_1 + \frac{\gamma_2}{\alpha_M - \alpha},$$

where the coefficients γ_1 and γ_2 are chosen such that $\phi(\alpha_j) = f'(\alpha_j)$ and $\phi'(\alpha_j) = f''(\alpha_j)$. The zero of this function is at $\alpha_\phi = \alpha_M + \gamma_2/\gamma_1$. If $\alpha_\phi \in I_j$, the new estimate is chosen as $\alpha_{j+1} = \alpha_\phi$; otherwise, repeated bisection is used on I_j until a midpoint α_{j+1} is found, such that $|f'(\alpha_{j+1})| < \min\{|f'(\underline{\alpha}_j)|, |f'(\overline{\alpha}_j)|\}$.

The first α_j to satisfy

$$0 \geq f'(\alpha_j) \geq \beta f'(0)$$

is chosen as the steplength α, where $\beta \in [0, 1)$ is a preassigned scalar. By restricting the choice to the α with $f'(\alpha) \leq 0$, we ensure a decrease of the objective function without evaluating it. This saves the effort of computing logarithms.

In practice, a close approximation to the minimum of $F(x + \alpha p)$ can be obtained after a small number (typically 1-3) of estimates α_j. Since the minimizer is usually very close to α_M, at least one variable will become very near to its bound if an accurate search is performed. Although this may sometimes be beneficial, the danger exists that the optimal value of that variable could be far from its bound. Thus, performing an accurate linesearch may temporarily degrade the speed of convergence. To guard against this, we use an upper bound of $0.98\alpha_M$ instead of α_M, and set $\beta = 0.9$.

Newton's method can be shown to have quadratic convergence in a neighborhood of the solution, provided the Hessian is not singular. In this neighborhood a step $\alpha = 1$ is taken. However, with the logarithmic barrier function this neighborhood is very small and generally decreasing with μ. Given the accuracies sought in solving the subproblems, this aspect of Newton's method is of little significance here.

Chapter 2

Beyond Nonnegativity

In practical problems, many variables are given bounds other than a lower bound of zero. This more general type of linear program can be solved by the barrier method without reformulation when the nonnegativity condition on x is replaced by

$$\ell \leq x \leq u.$$

Components of x can now be free variables, fixed variables or have any combination of upper and lower bounds, so that $\ell_j \in \Re \cup \{-\infty\}$ and $u_j \in \Re \cup \{\infty\}$ with $\ell \leq u$.

More Slack Variables

The ability to define fixed variables is utilized to specify a *slack variable* for every constraint. Typically in linear programming formulations an inequality constraint $a_i' x \leq b_i$ (with a_i' being a row of A) is converted to an equality constraint $a_i' x + x_{n+i} = b_i$ by introducing a slack variable x_{n+i}, such that $x_{n+i} \geq 0$. These slack variables do not appear in the objective function.

This concept is extended to the constraints that were originally in equality form, by requiring that $0 \leq x_{n+i} \leq 0$. These fixed slacks are introduced in order to make sure that the constraint matrix A is of full row rank, regardless of possible redundancies or degeneracy in the original formulation. The corresponding entry h_{n+i} of the Hessian is not defined, but we can set $h_{n+i}^{-1} = 0$. (See also page 34 for an extension to these definitions.)

The General Problem

To summarize the extensions to the SLP of Chapter 1 let us update or refine some of the definitions. The variables x are partitioned into a variable and a slack part, using the

notation

$$x = \begin{pmatrix} x_N \\ x_M \end{pmatrix} \quad \text{and similarly} \quad c = \begin{pmatrix} c_N \\ 0 \end{pmatrix}, \quad A = \begin{pmatrix} A_N & I \end{pmatrix},$$

where $x_N, c_N \in \Re^n$, $x_M \in \Re^m$, $A_N \in \Re^{m \times n}$ and I is the identity of dimension m. Uppercase subscripts denote partitions of vectors or matrices, while N and M were chosen here to reflect the dimensions n and m.

The general linear programming problem solved by our algorithm is of the form

$$\begin{array}{lll} \text{GLP} & \underset{x}{\text{minimize}} & c^T x \\ & \text{subject to} & Ax = b \\ & & \ell \leq x \leq u. \end{array}$$

Let $y = x - \ell$ and $z = u - x$ be the distances of x from its lower and upper bounds, respectively. The k-th logarithmic barrier subproblem is generalized to be

$$\begin{array}{ll} \underset{x}{\text{minimize}} & F^k(x) = c^T x - \mu^k \sum_j (\ln y_j + \ln z_j) \\ \text{subject to} & Ax = b, \end{array}$$

and the derivatives of $F(x)$ are given by

$$g = c + g_B, \quad g_{B_j} = -\mu(1/y_j - 1/z_j)$$

and

$$H = \operatorname{diag} h, \quad h_j = \mu(1/y_j^2 + 1/z_j^2), \quad j = 1, \ldots, m+n.$$

Note that these derivatives are well defined — even when a variable x_j is not bounded above or not bounded below. In these cases we use $1/y_j = 0$ for $\ell_j = -\infty$, and $1/z_j = 0$ for $u_j = \infty$.

Fixed Variables

When a set of related linear programs is solved, it is sometimes interesting to change the range of a variable, and in the extreme case, fix it to a certain value. Since the iterates of the barrier method need to be interior to the feasible region, fixed variables must be removed from the problem. Let x be partitioned into a fixed part x_F and a variable part x_V, so that corresponding partitions of the bounds satisfy $l_F = u_F$ and $l_V < u_V$. With the analogous

partition $A = (A_V\ A_F)$, one approach is to reduce the LP problem to

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c_V^T x_V \\ \text{subject to} \quad & A_V x_V = b - A_F \ell_F \\ & \ell_V \le x_V \le u_V, \end{aligned}$$

where the objective function differs from the original by the constant $c_F^T \ell_F$.

Arithmetically equivalent is an approach that treats $l_j = u_j$ as the limiting case of $l_j < u_j'$ with $u_j' \to l_j$. As $u_j' \to l_j$ we have $h_j^{-1} = h_j^{-1}(u_j') \to 0$, so that the corresponding entry on the diagonal of the inverse of the Hessian vanishes. Partitioning the system of the normal equations accordingly we see that

$$AH^{-1}A^T\pi = (A_V H_V^{-1} A_V^T + A_F 0 A_F^T)\pi = A_V H_V^{-1} A_V^T \pi = A_V H_V^{-1} g_V,$$

which are the normal equations for the reduced problem. At each iteration, the resulting multipliers π are therefore those of the reduced problem and the search direction is $p^T = (p_V^T\ 0)$.

When translated into an algorithm, however, these two approaches differ in one detail. With the reduced problem, the sparse factorization routine for the normal equations works on the matrix $A_V H_V^{-1} A_V^T$, whereas with the second approach, $AH^{-1}A^T$ is factorized. Although mathematically equivalent, the factorization of $A_V H_V^{-1} A_V^T$ can be expected to be more efficient than that of $AH^{-1}A^T$ (see Chapter 7 for the issues involved in sparse matrix algebra). However, the formulation that treats fixed variables as a limiting case, is interesting in that it offers the flexibility to fix (or free) variables dynamically for algorithmic reasons. This technique was used in [GMSTW86] but was not investigated further in this research.

(To preserve the full rank of A, fixed *slack* variables are not removed; see page 34. See the footnote on the bottom of page 26 for a discussion of multipliers for fixed variables.)

Free Variables: a Special Case

When $l_j = -\infty$ and $u_j = +\infty$ we call x_j a *free variable*. The corresponding entry on the diagonal of the Hessian is $h_j = \mu(1/(x_j - l_j)^2 + 1/(u_j - x_j)^2) = 0$ and h_j^{-1} does not exist. In this case the procedure to compute the search direction has to be reexamined. Let

$x^T = (x_b^T \; x_f^T)$ be a partition of x into its free and bounded parts and let A, p, c, g and H be partitioned accordingly. The KKT system is of the form

$$\begin{pmatrix} H_b & 0 & A_b^T \\ 0 & 0 & A_f^T \\ A_b & A_f & 0 \end{pmatrix} \begin{pmatrix} -p_b \\ -p_f \\ \pi \end{pmatrix} = \begin{pmatrix} g_b \\ c_f \\ 0 \end{pmatrix}.$$

Let D be a diagonal matrix such that $D^2 = H_b^{-1}$ and let $r = -D^{-1}p_b$. The system above may be rewritten as

$$\begin{pmatrix} I & 0 & DA_b^T \\ 0 & 0 & A_f^T \\ A_bD & A_f & 0 \end{pmatrix} \begin{pmatrix} r \\ -p_f \\ \pi \end{pmatrix} = \begin{pmatrix} Dg_b \\ c_f \\ 0 \end{pmatrix}.$$

As in the general least-squares formulation of page 9, the vector π in this equation is the minimizer of the *constrained* least-squares problem

$$\begin{aligned} \underset{\pi}{\text{minimize}} \quad & \|D(g_b - A_b^T\pi)\|_2^2 \\ \text{subject to} \quad & A_f^T\pi = c_f. \end{aligned}$$

Let ψ denote multipliers associated with the equality constraints $A_f^T\pi = c_f$. The gradient of the Lagrangian of the constrained least-squares problem is of the form

$$L'(\pi, \psi) = A_bH_b^{-1}A_b^T\pi - A_bH_b^{-1}g_b - A_f\psi.$$

(The factor 2 was dropped here.) With $B = (A_bH_b^{-1}A_b^T)^{-1}$ the solution is $\pi = BA_bH_b^{-1}g_b + BA_f\psi$. Since π has to satisfy $A_f^T\pi = c_f$, the multipliers ψ are

$$\psi = (A_f^TBA_f)^{-1}(c_f - A_f^TBA_bH_b^{-1}g_b).$$

Consequently π is the solution of the system

$$A_bH_b^{-1}A_b^T\pi = A_bH_b^{-1}g_b + A_f(A_f^TBA_f)^{-1}(c_f - A_f^TBA_bH_b^{-1}g_b).$$

Note that this formula reduces to the normal equations $A_bH_b^{-1}A_b^T\pi = A_bH_b^{-1}g_b$ in the case where all variables are bounded. Vanderbei [Van89] arrives at the same result for the affine-scaling algorithm by treating free variables as the limiting case of $-u_j \le x_j \le u_j$ with $u_j \to \infty$.

This approach has computational disadvantages. The matrix $A_f^TBA_f$ and its factors must be treated as dense, even for a sparse A_f. This would be inefficient for anything

but a small number of free columns. Also, because free variables are generally basic in the solution, $A_b H_b^{-1} A_b^T$ is more likely to be singular or ill-conditioned than $AH^{-1}A^T$. [2]

Instead of solving the constrained least-squares problem exactly, the following unconstrained penalty function (see, e.g. [VLo85]) is minimized to avoid these disadvantages:

$$\underset{\pi}{\text{minimize}} \|D(g_b - A_b^T\pi)\|_2^2 + \rho^2\|c_f - A_f^T\pi\|_2^2,$$

where ρ is a positive scalar. Denoting the solution of the approximate problem by $\pi(\rho)$, it can be shown that $\pi(\rho) \to \pi$ as $\rho \to \infty$.

Solving this unconstrained problem is equivalent to solving a KKT system in which $H_f = 0$ is approximated by $H_f = 1/\rho\, I$. Since $g_f = c_f$, this corresponds to approximating the infinite bounds of x_f by two equidistant bounds $l_f = x_f - \sqrt{2\mu\rho}\ \mathbf{1}$ and $u_f = x_f + \sqrt{2\mu\rho}\ \mathbf{1}$. These bounds are reset at every iteration and they are artificial, not only in the sense that they are not part of the original problem, but also that they are not used when determining the maximum feasible step along the search direction. The equivalence with the approximated least-squares problem ensures the convergence of this approach.

Observe that $p_f = -\rho(c_f - A_f^T\pi(\rho))$, where we can assume that the estimate of the Lagrange multiplier π is nearly constant in ρ for large ρ. Since the maximum stepsize α with $x + \alpha p$ feasible is independent of the size of p_f the change in the free variables $\|\alpha p_f\|$ is increasing in ρ. A small value of ρ can therefore impede rapid convergence, especially during early iterations or for unscaled problems. Conversely, a large value increases the ill-conditioning of the problem (see page 34).

[2] A similar problem exists for dense colums of A. They are taken out of the (main) Cholesky factorization as suggested here for the columns of A_f. The issue of efficiency is different for dense columns, since a great amount of computational work is saved by doing so (see Chapter 8). This suggests that solving the constrained least-squares problem exactly bears some promise in the case where columns in A_f are dense. In particular this is true for the artificial column (see [Van89] and page 22).

Chapter 3

Getting Started

The algorithm as stated requires a *strictly* feasible (or interior) initial point. In general, such a point can not be trivially determined.

One way to find a point that is feasible, though not necessarily interior, is to solve an augmented linear program (ALP). The LP is augmented in the sense that an artificial variable x_a and a corresponding column of the constraint matrix is added, making any starting point x^0 with $l \leq x^0 \leq u$ feasible. Let $a_{\text{inf}} = b - Ax^0$ be the vector of infeasibilities and let $a = \|a_{\text{inf}}\|^{-1} a_{\text{inf}}$ be the normalized version of this vector; then we solve

$$\begin{array}{lll} \text{ALP} & \underset{x, x_a}{\text{minimize}} & x_a + \omega c^T x \\ & \text{subject to} & Ax + a x_a = b \\ & & l \leq x \leq u \\ & & x_a \geq 0, \end{array}$$

where ω is a nonnegative weight. The artificial variable $x_a \in \Re$ is initialized to $x_a^0 = \|a_{\text{inf}}\| \geq 0$, so that (x^0, x_a^0) is feasible for ALP.

Depending on whether ω is positive or zero, the solution of ALP is an optimal or just a feasible point for the original LP. Although this approach seems straightforward, there are difficulties, some in general and some specific to an interior-point method. In this chapter we shall explore: (1) the implications of the choice of ω; (2) better bounds for x_a; and (3) what comprises a good starting point x^0.

The Meaning of the Weight ω

For $\omega = 0$ this scheme has two phases. In Phase I, the *feasibility phase*, we solve ALP and obtain a feasible point for the original LP. This is taken to be the starting point for Phase II,

the *optimality phase*, which solves the original LP for an optimal solution. The case where no solution with $x_a = 0$ can be found during Phase I, indicates an empty feasible region.

When $\omega > 0$, we say that ALP has a *composite objective function.* This approach can be seen as a variant with overlapping feasibility and optimality phases. More cases have to be considered for this variant and their interpretation has some ambiguity. If the algorithm successfully terminates and $x_a = 0$, the solution vector x is not only feasible but also optimal to the original LP. If the objective function is unbounded below, but $x_a = 0$ for all points on the unbounded feasible direction, the original problem is unbounded. However if there exists a solution or a feasible direction, respectively, with $x_a > 0$, either the feasible region is empty or ω was chosen to be too large. For every linear program there exists a value ω' so that any augmented problem with $0 < \omega < \omega'$ has the same solution as the original problem. Unfortunately the determination of ω' is not easy, since it would require the solution of a nonlinear program of the same size as the original LP.

Let us examine the two-phase scheme ($\omega = 0$) in connection with the barrier method. Under certain regularity conditions, the solution found by the barrier method in Phase I is not only feasible but also interior for the LP solved in Phase II. For simplicity, consider the linear program

$$\underset{x}{\text{minimize}}\ c^T x \qquad \text{subject to } Ax = b, \qquad x \geq 0,$$

and assume that its interior $\{x > 0 \mid Ax = b\}$ is non-empty and bounded. When the ALP

$$\underset{x,x_a}{\text{minimize}}\ x_a \qquad \text{subject to } Ax + ax_a = b, \qquad x \geq 0,\ x_a \geq 0$$

is solved by the logarithmic barrier method, the subproblems are of the form

$$\underset{x,x_a}{\text{minimize}}\ x_a - \mu(\ln x_a + \sum \ln x_j) \quad \text{subject to } Ax + ax_a = b.$$

Let $(x^*(\mu), x_a^*(\mu))$ be the solution of one barrier subproblem. The strict convexity of the objective function implies that $x^*(\mu)$ is also the unique optimal point of the problem

$$\underset{x}{\text{minimize}}\ -\mu \sum \ln x_j \qquad \text{subject to } Ax = b - ax_a^*(\mu),$$

which is formed by fixing the artificial variable at its optimal value. The limit of this sequence of problems as $\mu \to 0$ is

$$\underset{x}{\text{minimize}}\ -\sum \ln x_j \qquad \text{subject to } Ax = b,$$

since $x_a^*(\mu) \rightarrow 0$. The objective function of this last problem is only finite for $x > 0$. The solution $x^*(0)$ therefore lies in the interior of the feasible region, or could even be defined as its center. Consequently $x^*(0)$ is a feasible interior point for the original LP. The argument carries over for the case with upper and lower bounds. The assumption of a bounded feasible region can be dropped when the modified barrier function of page 24 is used.

Thus the two-phase method ($\omega = 0$) would be the method of choice if it were not for the fact that it has clear performance disadvantages compared to using the composite objective function. Generally speaking, information about the problem gathered in Phase I is lost when Phase II has a totally different objective function. More specifically, an approximate solution found by the barrier method for a problem with little or no interior, will have variables close to their bounds. This may be a bad starting point for the barrier method, especially if the close bounds are not active at the optimal solution of Phase II. It is therefore advantageous to have the solution of the feasibility phase coincide with that of the optimality phase.

Experiments show that the time for overall convergence improves with increasing ω in almost all cases. This implies that a good ω would be one close to ω'. A practical approach is to set ω initially to some *a priori* value that has performed reliably for a good range of problems in the past, and reduce it when no satisfactory reduction of x_a is achieved during the solution of one subproblem, say. Our tests showed satisfactory results with $\omega \in [0.0001, 1.0]$ for a normalized objective function, $\|c\| = 1$. The reduction requirement we impose is $x_a^k \leq \beta x_a^{k-1}$ with $\beta \in [0.5, 0.9]$.

Bounds on the Artificial Variable

Since we use an artificial column that is normalized, x_a *is the norm of infeasibilities* at every iteration. The nonnegativity constraint $x_a \geq 0$ reflects this nature of the artificial variable. However, using it as such in a barrier algorithm would make it impossible to find a feasible point in a finite time, since variables are barred from attaining their bounds. Consequently, we relax this bound to some sufficiently negative value, while ensuring that x_a never actually becomes negative.

Specifically, if a search direction p and a steplength α are chosen so that $x_a + \alpha p_a \leq 0$, then α is reduced to $\alpha = -x_a/p_a$. At this point the artificial column is removed from the problem and Phase II begins.

Note that this technical detail removes the structural difference between the cases with or without a positive weight ω, since it introduces a true optimality phase to the case with $\omega > 0$. This optimality phase will usually be short for a big ω, but there is still some speed advantage from the fact that one totally dense column, the artificial column a, is removed from the problem (see Chapter 8).

In order to remove the artificial column, an optimality phase is introduced even for problems that have no interior. For these problems, x_a approaches zero in the limit. Phase II is selected as soon as the infeasibility falls below some threshold value, i.e., $x_a \leq \epsilon_{\text{feas}} \|x\|$, where ϵ_{feas} is the accuracy to which we want to see the constraints $Ax = b$ satisfied. This tolerance cannot be smaller than the precision that can be attained when solving the (often ill-conditioned) systems towards the end. We chose $\epsilon_{\text{feas}} = 10^{-6}$ or 10^{-8} as a generally satisfactory standard.

Let us return to the question of formulating suitable bounds for x_a. Although a negative bound would never be active, the associated barrier term might still impede the convergence of x_a. Alternatively, we could impose an upper bound on x_a. This bound would be reset at the beginning of each subproblem to a value slightly larger than the present value of x_a so as to encourage some progress towards feasibility.

Such reasoning ignores a peculiarity of the logarithmic barrier function, namely that, given an α fixed by the bounds of other variables, the change in x_a will increase with its distance from a bound. If we assume for illustration purposes that the constraint matrix A is empty, then the Newton search direction can be readily computed as $p = -H^{-1}g$. Since x_a cannot be defined as the norm of infeasibilities under these conditions, let x_a be any variable with $c_a = 1$. If we impose an upper bound u_a, the element of p associated with x_a is $p_a = -z_a^2/\mu - z_a$ with $z_a = u_a - x_a$, or if we impose a lower bound l_a, it is $p_a = -y_a^2/\mu + y_a$ with $y_a = x_a - l_a$. This indicates that the change in x_a depends more on the distance to the bound than on whether it is an upper or lower bound, and that a close bound will yield a very small change.

Naturally things look different with equality constraints, but the tendency shown here is similar to the behavior of the algorithm observed in practice. Specifically, a lower bound $l_a = -1$ (with $u_a = \infty$) gives almost as good results as a dynamic upper bound $u_a^k = 2x_a^{k-1}$ (with $l_a = -\infty$), while both show much faster convergence to a feasible point than an upper bound $u_a^k = x_a^{k-1} + 1$ (with $l_a = -\infty$).

In summary, the artificial variable x_a is best treated as a free variable (see page 16).

We conclude this section with one more consideration of a numerical nature. It is not uncommon, especially with unscaled problems, to start with a point x^0 that has large

infeasibilities, so that $x_a^0 \approx \epsilon_{\text{feas}}/\epsilon_M$ (ϵ_M = machine precision). In these cases the rounding error in x_a^0 makes later comparisons of x_a with ϵ_{feas} meaningless. Additionally, errors are accumulated in x because of ill-conditiong in the systems that determine the sequence of search directions p. To guard against the accumulation of excessive error, the artificial column a and x_a are recomputed at the beginning of every subproblem.

Convergence is ensured by monitoring the reduction of the norm of infeasibilities x_a. If the reduction during one subproblem falls below a satisfactory value, the weight ω is adjusted downward.

Where to start

As with most iterative methods, the choice of the starting point for the barrier function method will have a great impact on the performance. What is special here is that any knowledge of an approximate solution does not necessarily improve efficiency. For example, starting off with a solution that was derived from the basis of a related LP, which is typically done with the simplex method, is usually undesirable. At such a starting point, several variables are very close to their bounds. If the new optimum is not near those bounds, this choice of a starting point results in slow convergence and possible ill-conditioning of the normal equations.

Experience shows that subproblem k converges most rapidly when started with the solution of subproblem $k-1$. The sequence of solutions $x^*(k)$ lies on the barrier trajectory. In order to start the algorithm on this trajectory, x^0 should be a good guess at the solution $x^*(0)$ of subproblem 0. This problem can be seen as a backward extrapolation of the sequence of subproblems $k = 1, 2, \ldots$ that are solved. One method to determine an $x^*(0)$ is to solve the unconstrained problem

$$\min\ F^0(x) = \omega c^T x + \sum_j f_j^0(x_j),$$

which uses the objective function of subproblem 0. The constraints $Ax^0 + ax_a^0 = b$ are then satisfied by setting x_a^0 and a according to their definitions. The unconstrained problem is separable and a solution, if it exists, would simply be the zeros of the elements of the gradient of F^0.

A solution does not exist or is of little use for the simple logarithmic barrier function, e.g. $f_j(x_j) = -\mu \ln(x_j - l_j)$ for lower bounded x_j. For $c_j \leq 0$ this function has no minimum and even for $c_j > 0$ the minimizer is given by $x_j = \mu^0/(\omega c_j)$, which may be large.

Linear Modification to the Barrier Function

A barrier function for which there exists a minimizer for every μ, is one that includes a linear term. Let ν be a small, positive scalar and let

$$f_j(x_j) = \mu(\ln y_j - \nu y_j), \quad y_j = x_j - l_j,$$

and

$$f_j(x_j) = \mu(\ln z_j - \nu z_j), \quad z_j = u_j - x_j,$$

define the barrier terms of page 7 for the lower and the upper bounded variables, respectively. (Note, for variables where both bounds l_j and u_j are finite, the linear terms form a constant $\nu z_j + \nu y_j = \nu(u_j - l_j)$ and can be eliminated from the minimization. The result is the simple function $f_j(x_j) = \mu(\ln y_j + \ln z_j)$.)

The minimizers of this barrier function for the lower and upper bounded variables are $x_j = l_j + \mu/(\mu\nu + \omega c_j)$ and $x_j = u_j - \mu/(\mu\nu - \omega c_j)$. When we choose ν such that $\mu^0\nu \gg \omega\|c\|$, we can disregard the linear part $\omega c^T x$ of the objective function. The elements of our starting point close to the trajectory are therefore

$$x_j^0 = l_j + 1/\nu, \qquad x_j^0 = u_j - 1/\nu \qquad \text{or} \qquad x_j^0 = (u_j + l_j)/2,$$

for the three kinds of bounded variables, respectively. Note that this approximation of the minimum of $F^0(x)$ can be given even without knowing μ^0 exactly. This is an advantage when μ^0 is chosen, for example, as a function of x_a.

The trajectory $x^*(\mu)$ of this modified barrier function differs significantly from the trajectory of the simple logarithmic barrier function in that it is bounded. In particular, the starting point x^0 satisfies

$$x^0 = \lim_{\mu \to \infty} x^*(\mu).$$

The consequence is that there is no danger of choosing μ^0 too big and thereby driving the iterates away from the solution.

Starting from the point x^0 as defined above, the algorithm achieves fast convergence for the first few subproblems for a wide range of linear programs. The parameters ν used were in the range $[10^{-5}, 10^{-1}]$. Larger values tend to give better results for scaled problems, but are less reliable for unscaled problems.

There is some degree of freedom in choosing x^0, since the objective function $F(x)$ is relatively insensitive to changes in x in a neighborhood of its minimizer. Additional time savings were obtained when each x_j^0 was chosen in a neighborhood of the value above so as to reduce the infeasibilities in x^0.

Bounding the Optimal Region

It should be noted that, apart from helping to find a good starting point, the linear modification of a barrier objective function is essential for solving a rare class of problems. These are problems where the set of optimal points is unbounded.

Let x^* be a solution of an LP that lies at a vertex, and let d be a feasible direction with $Ad = 0$ and $l \leq x^* + \alpha d \leq u$ for all $\alpha \geq 0$. Since x^* is optimal we know that $c^T d \geq 0$. If there exists a d such that $c^T d = 0$, the barrier function subproblem does not converge since the barrier function is strictly decreasing in α in that direction, i.e., $\|x^*(\mu)\| \rightarrow \infty$. The linear modification ensures convergence to a finite minimum in that case.

Chapter 4

Where to Stop

Several references have been made so far to the *solutions* $x^*(k)$ of the subproblems and the *solution* x^* of the original linear program. Since both major iterations (subproblems) and minor iterations (Newton steps) converge in the limit, we must define the point at which we accept the current iterate as the solution.

A number of properties of an iterate x indicate its closeness to a solution. We shall review these properties in this chapter, first for the general LP and later for the barrier subproblems. Later we shall examine the relationship between convergence criteria and the barrier parameters μ^k.

Complementarity

As before, let $y = x - l$ and $z = u - x$ be the distances of x from its bounds. The Lagrangian of the GLP of page 15 is

$$L(x, \pi_L, \eta_l, \eta_u) = c^T x - \pi_L^T (Ax - b) - \eta_l^T y - \eta_u^T z,$$

where π_L are the multipliers for the equality constraints $Ax = b$ and η_l, η_u are those for the lower and upper bounds.[3]

[3] There is some interest in computing the multipliers η_F for fixed variables x_F, where $\ell_F = u_F$ (see page 15 for notation used here). These can be calculated from the optimality conditions as $\eta_F = c_F - A_F^T \pi_L$ and correspond to the multipliers of the equality constraints $Ix_F = \ell_F$, had they been used to define fixed variables instead of $\ell_F \le x_F \le u_F$. To see that, let $\bar{A}$ be the augmented constraint matrix containing these equality constraints, and observe

$$\bar{A}^T \begin{pmatrix} \eta_F \\ \pi_L \end{pmatrix} = \begin{pmatrix} I & A_F^T \\ 0 & A_V^T \end{pmatrix} \begin{pmatrix} \eta_F \\ \pi_L \end{pmatrix} = \begin{pmatrix} \eta_F \\ 0 \end{pmatrix} + A^T \pi_L = \begin{pmatrix} c_F \\ c_V \end{pmatrix} - \begin{pmatrix} 0 \\ \eta_l - \eta_u \end{pmatrix}.$$

These multipliers are independent of whether the fixed variables were explicitly excluded from the problem or not.

Necessary conditions for optimality are

$$\nabla_x L \equiv c - A^T \pi_L - (\eta_l - \eta_u) = 0,$$

together with the nonnegativity and complementarity conditions

$$\eta_l \geq 0, \quad \eta_l^T y = 0$$
$$\eta_u \geq 0, \quad \eta_u^T z = 0.$$

(Throughout this discussion we assume that $\eta_{lj} = 0$ whenever $l_j = -\infty$ and define $\eta_{lj}\, y_j \equiv 0$ in this case. The equivalent holds for $u_j = \infty$.)

Let $\pi_L(x)$, $\eta_l(x)$ and $\eta_u(x)$ be suitable estimators of the multipliers corresponding to the current iterate x, so that $\eta_l(x) \geq 0$, $\eta_u(x) \geq 0$ and $c - A^T\pi_L(x) - (\eta_l(x) - \eta_u(x)) = 0$. If we add the condition that $\eta_{lj}(x) \to 0$ if $x_j \to u_j$ and $\eta_{uj}(x) \to 0$ if $x_j \to l_j$, we can estimate the sum of complementarity violations

$$s = \eta_l^T(x) y + \eta_u^T(x) z.$$

The scalar s is an indicator of convergence, since $s \to 0$ for $x \to x^*$ and $s > 0$ for every x that is not a solution of the LP.

To derive meaningful estimators, let us recall from page 8 the other two optimality conditions based on gradients of Lagrangians: for the barrier subproblem,

$$g - A^T \pi_b = 0,$$

and for the quadratic program solved at every (minor) iteration,

$$g + Hp - A^T \pi = 0.$$

Since $\pi \to \pi_b$ as $x \to x^*(k)$ for each subproblem and $\pi_b \to \pi_L$ as $\mu^k \to 0$ for the sequence of subproblems, we use $\pi_L(x) = \pi$ as the estimator of the equality-constraint multipliers. This implies that $\eta_{l_j}(x) = c_j - a_j^T\pi$ for $u_j = \infty$, and $\eta_{uj}(x) = -c_j + a_j^T\pi$ for $l_j = -\infty$. For the case where both bounds are finite there is some degree of freedom in finding estimators. One possible definition is

$$\eta_{lj}(x) = \frac{z_j}{u_j - l_j}(c_j - a_j^T\pi) \quad \text{and} \quad \eta_{uj}(x) = \frac{y_j}{u_j - l_j}(c_j - a_j^T\pi).$$

These estimators are nonnegative (i.e., useful) only when the primal iterate x is sufficiently close to the solution.

Duality Gap

A related idea is based on duality theory. The dual of the GLP (page 15) is of the form

$$\begin{aligned} \underset{\pi_L, \eta_l, \eta_u}{\text{maximize}} \quad & F_D = b^T \pi_L + l^T \eta_l - u^T \eta_u \\ \text{subject to} \quad & A^T \pi_L + \eta_l - \eta_u = c \\ & \eta_l, \eta_u \geq 0. \end{aligned}$$

A standard result from duality is that $F_D \leq c^T x$ for all primal feasible and dual feasible points, and that $F_D^* = c^T x^*$, where F_D^* is the optimal objective function value of the dual.

Using the same estimators for the dual variables as defined in the last section, an estimate $F_D(x)$ of the dual objective function can be computed. The relative difference between the two objective functions, namely

$$d = \frac{c^T x - F_D(x)}{|c^T x| + |F_D(x)| + 1},$$

is another indicator of convergence, since $d \to 0$ as $x \to x^*$ and $d > 0$ for every x that is not a solution of the LP.

Termination of a Subproblem

The solution $x^*(k)$ of subproblem k is not interesting as such, except as the starting point for the subproblem $k+1$. There is little need to seek a highly accurate approximation of $x^*(k)$, since a point near the barrier trajectory should be satisfactory. It is for this reason that the quadratic convergence of Newton's method in a small neighborhood of the solution is of little significance.

Three vectors tend to zero in Newton's method as $x \to x^*(k)$, namely the search direction p, the estimate of the gradient of the Lagrangian $g_L = g - A^T \pi$, and the *reduced gradient* $r = \sqrt{H^{-1}} g_L$, which is the optimal residual of the least-squares problem on page 9. All are diagonal scalings of each other, since $g_L = \sqrt{H} r = Hp$ (see page 8).

Each of these three quantities could serve as an indicator for the degree of convergence achieved so far. During testing it was observed that $\|r\|$ was the most consistent and reliable measure of convergence.

Convergence and the Barrier Parameter

The accuracy required for a given subproblem is a function of the barrier parameter μ^k. Barrier subproblems with small values of μ benefit more from a starting point close to the trajectory. Only the last subproblem need be solved to the accuracy required in x^*.

The algorithm that controls the convergence of the subproblems is of the following form. For subproblem k, a target level $\bar{r}^k$ for the norm of the reduced gradient is computed as a fraction of the final norm $\|r^{k-1}\|$ from the previous subproblem, i.e., $\bar{r}^k = \phi_r \|r^{k-1}\|$. As soon as $\|r\| < \bar{r}^k$, a new subproblem is started with $\mu^{k+1} = \phi_\mu \mu^k$ and a new target level $\bar{r}^{k+1}$ is determined. The reduction factors ϕ_r and ϕ_μ must lie in the interval $(0,1)$ to be meaningful. In the final subproblem, where $\mu^k = \mu_{\min}$, the level is set to a predefined minimum $\bar{r}_{\min}$.

In contrast to a test on $\|r\|$, the convergence criteria based on the complementarity violation s or the duality gap d cannot be employed during early subproblems. At the beginning, the estimates of the dual variables are inaccurate or not dual feasible, and $d \approx 1$ as long as the objective function of the primal and the dual problem have different signs. These criteria can be used to supplement a criterion based on $\|r\|$ during the last subproblem. In our implementation, the reduced gradient is the only indicator of convergence used.

The values of the reduction factors determine the number of the subproblems and the time it takes to solve each. The values used in the tests were $\phi_\mu = 0.1$ and $\phi_r = 0.1$. The behavior of the algorithm is surprisingly independent of the starting value μ^1. Values tested were $\mu^1 \in (10^{-4}, 1)$ and $\mu_{\min} = 10^{-6}$, both multiplied by $c^T x/n$.

Part II Computing the Search Direction

Chapter 5

The Toolkit

In Part II we shall explore different aspects of solving the KKT-system

$$\begin{pmatrix} H & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} -p \\ \pi \end{pmatrix} = \begin{pmatrix} g \\ 0 \end{pmatrix},$$

for the current estimate π of the multipliers and a search direction p.

Premultiplying the first part $-Hp + A^T\pi = g$ by AH^{-1}, we derive the *normal equations*

$$AH^{-1}A^T\pi = AH^{-1}g.$$

The matrix $AH^{-1}A^T$ is symmetric and positive definite. Since A is of the form $A = (A_N\ I)$, let $H = \text{diag}(H_N, H_M)$ be partitioned accordingly. The nonzero structure of the product $AH^{-1}A^T = A_N H_N^{-1} A_N^T + H_M^{-1}$ can be seen to be that of $A_N A_N^T$. The efficiency of recent methods for forming the triangular Cholesky factors $AH^{-1}A^T = LL^T$ (see [GL81,87]) has given the normal equations a prominent role in the implementation of interior-point algorithms.

Before going into the details and potential hazards of this approach in the following chapters, we review some alternatives and their characteristics.

The Least-Squares Problem

In Chapter 1 we mentioned that the term "normal equations" is derived from the weighted least-squares problem (page 9)

$$\underset{\pi}{\text{minimize}}\ \|Dg - DA^T\pi\|_2^2,$$

which is equivalent to the KKT-system with $D^2 = H^{-1}$.

However, there are other ways of solving large sparse least-squares problems. Three of these methods, two direct and one iterative, are described below.

The QR Factorization

Let $C = DA^T$ be the matrix associated with the least-squares problem. There exist an orthonormal matrix Q and a factor R such that

$$C = QR = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R_1 \\ 0 \end{pmatrix},$$

where R_1 is square and upper triangular. Since the Euclidean length of a vector is invariant under an orthogonal transformation we can rewrite the norm of the least-squares problem as

$$\|Dg - C\pi\|^2 = \|Q^T Dg - Q^T C\pi\|^2 = \|Q^T Dg - R\pi\|^2 = \|Q_1 Dg - R_1\pi\|^2 + \|Q_2 Dg\|^2,$$

so that the optimal π is the result of the backward substitution $R_1\pi = Q_1 Dg$.

Strong error bounds can be derived for the QR factorization in finite-precision arithmetic (see [GVL83]), which makes it more desirable than the Cholesky factorization in terms of numerical stability.

The disadvantage of the QR factorization is its computational cost. The number of operations involved in a sparse QR factorization is considerably larger than that of a Cholesky factorization of $AH^{-1}A^T$, especially when A is very rectangular. (See [GN84] and [GLN88] for implementations of sparse QR. The matrix Q need not be stored in our case, but stands for a series of orthogonal transformations applied at the same time to C and Dg.)

Given what we know about the QR factorization today, we do not expect interior-point methods based on this factorization to be competitive.

The Semi-Normal Equations

Note that the Cholesky factors of $AH^{-1}A^T$ are related to the QR factors of C. We have

$$LL^T = AH^{-1}A^T = C^T C = R^T Q^T Q R = R^T R = R_1^T R_1.$$

Thus L can be computed by performing the QR factorization and setting $L = R_1^T$. The method of semi-normal equations consists of forming L this way and solving for π with the normal equations $LL^T\pi = AH^{-1}g$.

The numerical properties of this method are analyzed in [Bj87a]. Although the triangular factor is of better "numerical quality", the error in π is shown to be about the same as that obtained from Cholesky factorization. The only improvement is in the bound on the condition number of C to achieve a numerically non-singular L. The concern of computational inefficiency with the QR factorization applies as before.

The Conjugate-Gradient Method

One algorithm for solving the least-squares problem that is not based on a matrix factorization but on a series of matrix-vector products, is the conjugate-gradient (CG) method. Starting at an initial point π_0, the method proceeds by taking steps along a sequence of search directions u_k. With initial values $r_0 = Dg$ for the residual, $u_1 = s_0 = C^T Dg$ and $\gamma_0 = \|s_0\|^2$, each iteration includes the following steps for $k = 1, 2, \ldots$:

$$\begin{aligned} q_k &= Cu_k \\ \alpha_k &= \gamma_{k-1}/\|q_k\|^2 \\ \pi_k &= \pi_{k-1} + \alpha_k u_k \\ r_k &= r_{k-1} - \alpha_k q_k \\ s_k &= C^T r_k \\ \gamma_k &= \|s_k\|^2 \\ \beta_k &= \gamma_k/\gamma_{k-1} \\ u_{k+1} &= s_k + \beta_k u_k. \end{aligned}$$

Certain orthogonality relations can be shown (see e.g. [HS52]); in particular, $s_k^T u_j = 0$, $s_k^T s_j = 0$ and $u_k^T C^T C u_j = 0$ for $j = 1, \ldots, k-1$.

In theory, this procedure can be considered a direct method since it converges in a number of iterations that is equal to the number of distinct singular values of C. In practice, rounding errors cause the algorithm to behave like an iterative method, and termination may occur whenever $\|s_k\|$ is sufficiently small. It is still observed to perform best on problems where the singular values of C are clustered in groups.

Variants of the conjugate-gradient method have been used successfully in implementations of interior-point methods, see [GMSTW86], [KR88]. The version used in [GMSTW86] is `LSQR` by Paige and Saunders [PS82] which is very well suited for solving least-squares problems. Other CG methods solve a system of the form $Bx = y$ and can be applied to the normal equations. Some vector operations may be saved that way, but it has much less desirable numerical properties.

The matrix C may be transformed into a matrix with clustered singular values by using a *preconditioner*. Let R be the nonsingular Cholesky factor of a matrix that approximates $C^T C$. The problem

$$\underset{z}{\text{minimize}} \|Dg - CR^{-1}z\|_2^2$$

can usually be solved using CG in fewer iterations. The original solution is recovered by solving $R\pi = z$.

At each CG iteration the main work is in forming products of the form $CR^{-1}u$ and $(CR^{-1})^T v$. The savings obtained by factorizing an approximation of C^TC compared to factorizing the exact matrix, have to be large enough to offset the cost of the iterations. The success of this approach lies entirely in the ability to devise a sparse preconditioner R such that R^TR has eigenvalues close to those of C^TC.

The Nullspace Method

An alternative approach to solving for π first is one based on the observation that p lies in the nullspace of A. Let Z be a matrix whose columns span the nullspace of A, so that $AZ = 0$ and for every p with $Ap = 0$ there exists a linear combination p_z of the columns of Z such that $p = Zp_z$. The first part

$$-Hp + A^T\pi = g$$

of the KKT-system is premultiplied by Z^T to give

$$Z^THZp_z = -Z^Tg.$$

As before, this system is symmetric and positive definite. It can be solved either directly by forming Cholesky factors, or by applying one of the previously discussed methods to the least-squares problem

$$\underset{p_z}{\text{minimize}} \|Dg - D^{-1}Zp_z\|_2^2.$$

For the special structure of $A = (A_N \;\; I_m)$, a matrix whose columns span the nullspace is given by

$$Z = \begin{pmatrix} -I_n \\ A_N \end{pmatrix}.$$

Observe that the sparsity structure of the positive-definite system $Z^THZ = H_N + A_N^TH_MA_N$ is that of $A_N^TA_N$. Since most linear programming problems have more dense rows than dense columns, this matrix is likely to have more nonzero elements (and hence be harder to factorize) than one of the form $A_NA_N^T$. (For more on the issue of comparing these sparsity structures, see page 54.)

Chapter 6

Ill-conditioned Systems

It is common for the matrix $AH^{-1}A^T$ of the normal equations to have a high condition number. The ill-conditioning may arise because A and/or H^{-1} are ill-conditioned.

The matrix $AH^{-1}A^T = A_N H_N^{-1} A_N^T + H_M^{-1}$ is near singular or singular when A_N is ill-conditioned and the diagonal of H_M^{-1} has some zero entries.[4] This is due to degeneracy in the formulation of the original problem. To detect degenerate rows that are redundant is a hard combinatorial problem. In addition, near rank-deficiency is likely to occur in problems that are poorly scaled.

Near-singularities occur also if the number of diagonal entries in H^{-1} approaching zero is greater than n, which is a typical behavior towards the end of Phase I or II when many variables are approaching their bounds.

More Slack for Fixed Slacks

The problem of a nearly rank-deficient A can be eased by introducing small bounds on the fixed slacks x_S of rows that were originally equality constraints, i.e., the constraints $0 \leq x_S \leq 0$ are replaced by $-\delta\, 1 \leq x_S \leq \delta\, 1$ with $\delta \geq 0$.

When $\delta > 0$, all diagonal entries of H_M^{-1} are nonzero and $AH^{-1}A^T$ is strictly positive definite. At the same time, the dimensionality of the feasible region is increased, possibly creating a strictly interior region. The parameter μ_s of a barrier function associated with these bounds is to be treated differently. Reducing μ_s from one subproblem to the next, does not help the convergence of the subproblems to the original problem. We would therefore like to keep μ_s constant and big, say $\mu_s \approx 10^5$, in order to ensure that $\mu_s > \mu^k$ for any k.

[4] We continue to use H_M^{-1} as a symbol, even if it is singular and H_M is not defined. This case may be viewed as the limit of shrinking bounds on the fixed slack variables.

Naturally, such bounds affect the precision with which the original constraints are satisfied at the solution. Let x_{n+i} be an entry in x_S and π_{L_i} the multiplier of the corresponding constraint at the solution. Since $g = A^T\pi_L$, it follows that

$$-\mu_s\left(\frac{1}{x^*_{n+i}+\delta} - \frac{1}{\delta - x^*_{n+i}}\right) = \pi_{L_i} \quad \text{or} \quad x^*_{n+i} = \frac{-\mu_s \pm \sqrt{\mu_s^2 + \delta^2\pi_{L_i}^2}}{\pi_{L_i}}.$$

Assuming that $|\pi_{L_i}| \ll \mu_s/\delta$, the value x^*_{n+i} can be approximated by $(\delta^2\pi_{L_i})/(2\mu_s)$. Since $|\pi_{L_i}| < 10^7$ for all but the worst scaled problems, a bound $\delta = 10^{-4}$ yields a solution that satisfies the feasibility tolerance $\epsilon_{\text{feas}} = 10^{-6}$.

As far as $AH^{-1}A^T$ is concerned, introducing no bounds on the fixed slacks, i.e. setting $\delta = 0$, is equivalent to removing the corresponding columns from A. Tests with scaled problems showed that this reduced constraint matrix was sufficiently well-conditioned in all but a few cases. It was also observed that the performance of the algorithm on other problems was degraded by introducing artificial bounds on fixed slack variables. In our implementation, δ is therefore a user-selectable parameter with a default value of zero. It has to be set to a positive value for problems where difficulties caused by the rank of A are encountered, and it can be reset to a smaller value when the resulting residual $\|Ax - b\|$ is deemed too large.

A Theoretical Bound

Concerning the difficulties introduced by an ill-conditioned H^{-1}, Dikin [Dik67] and Stewart [Stew87] show for a full-rank A that

$$\sup_{H \in \mathcal{D}^+} \|(AH^{-1}A^T)^{-1}AH^{-1}\| < \infty.$$

The set $\mathcal{D}^+$ is the space of diagonal matrices with positive diagonal elements. Since $\pi = (AH^{-1}A^T)^{-1}AH^{-1}g$ and $H \in \mathcal{D}^+$ by its definition, we should expect from this result that the numerical error in π is also bounded. However, short of using a QR factorization, we do not know how to form the matrix $(AH^{-1}A^T)^{-1}AH^{-1}$ without forming $(AH^{-1}A^T)^{-1}$ first (i.e. forming and factorizing $AH^{-1}A^T$). Since $\|(AH^{-1}A^T)^{-1}\|$ cannot be bounded on $\mathcal{D}^+$, the error has already been introduced at this point. The following are measures to improve the accuracy of π and to reduce the condition number of $AH^{-1}A^T$.

Updating π

When computing the π of one (minor) iteration, a fairly good estimate is already available in the form of the multiplier estimate $\bar{\pi}$ of the previous iteration. This is especially true towards the end of a barrier subproblem when $\pi \rightarrow \pi^*(\mu)$. In order to avoid rounding errors and to reduce the impact of catastrophic cancellations, the change $q \equiv \pi - \bar{\pi}$ is computed rather than π itself.

Therefore, the system solved to determine a new search direction p is of the form

$$\begin{aligned} AH^{-1}A^T q &= AH^{-1}\bar{g}_L \\ p &= -H^{-1}g_L, \end{aligned}$$

with $\bar{g}_L = g - A^T\bar{\pi}$ and $g_L = \bar{g}_L - A^T q = g - A^T\pi$. The vector is denoted by g_L because it converges to the gradient of the Lagrangian (page 8).

Diagonal Correction

The computed search direction p must satisfy two conditions. First, it must be close to the null space of A, which means that $\|Ap\|/\|p\| < \epsilon$ *for some suitable* $\epsilon > 0$. *Second, it must* be a descent direction for the barrier objective function, i.e., $g^T p < 0$. These conditions are satisfied as long as q is an exact solution for the system above, since

$$Ap = -AH^{-1}\bar{g}_L + AH^{-1}A^T q = 0$$

and

$$g^T p = (g_L + A^T\pi)^T p = -{g_L}^T H^{-1} g_L + \pi^T A p < 0$$

for any feasible, non-optimal point (x, π). Observe that $g^T p = -{g_L}^T H^{-1} g_L$ is less than zero if $Ap = 0$, independently of the accuracy in π. We can therefore focus on Ap as the error term in question.

In order to model the error introduced into q, assume q to be the exact solution of the system

$$(AH^{-1}A^T + E)q = AH^{-1}\bar{g}_L,$$

where E is an error matrix. The error term is then

$$Ap = Eq.$$

The error matrix E is small except for matrices $AH^{-1}A^T$ that are very ill-conditioned. In this small neighborhood of singularity there also exists some danger that the Cholesky factorization might break down because of diagonal elements that become extremely small or negative due to rounding error. One way to guard against a break-down or the large E associated with very ill-conditioned matrices is to add a correction matrix F to $AH^{-1}A^T$ that improves its condition number. Let $E(F)$ be the error matrix associated with the system $AH^{-1}A^T + F$. Then F should be chosen so that $\|F + E(F)\|$ is minimized, which means, F should be the smallest correction that brings $AH^{-1}A^T + F$ out of the neighborhood of singularity.

A good and simple choice for F is a diagonal matrix that reduces the quotient $l_{\max}/l_{\min}$ of the largest and smallest diagonal elements of the factor L. This heuristic is based on the fact that the condition number of $AH^{-1}A^T$ is bounded below by $(l_{\max}/l_{\min})^2$.

The correction matrix F may be formed during the factorization, by using all zero entries except for those indices i where the diagonal of L is below some threshold value, i.e.,

$$F_{ii} = (\max\{\gamma\, l_{\max} - L_{ii},\, 0\})^2,$$

for some $0 < \gamma \ll 1$. This definition is used in our implementation. The correction F_{ii} is computed at the point where L_{ii} is determined during the factorization. Such a choice for F has the advantage that F is zero for well-conditioned systems and relatively small otherwise, and that a bound $l_{\max}/l_{\min} \leq 1/\gamma$ is enforced. Nevertheless, examples of near-singular $AH^{-1}A^T$ can be constructed, where the correction can grow to $\|F\| = 2^m \epsilon_M$ (ϵ_M = machine precision). Since the exact $l_{\max}$ is not known until the factorization is complete, we use an estimate for determining F_{ii}. The estimate is $l_{\max}(i) = \max\{0.1\, l'_{\max},\, L_{jj} \text{ for } j = 1, \ldots, i-1\}$, where $l'_{\max}$ is the maximum of the previous iteration. In tests we used a threshold factor $\gamma = 0.1\sqrt{\epsilon_M}$.

Modified Hessian

We would expect to be able to improve on the error term by taking the correction F to the diagonal of $AH^{-1}A^T$ into account when subsequently computing p. Since

$$AH^{-1}A^T + F = A_N H_N^{-1} A_N^T \quad + \quad H_M^{-1} + F,$$

where both H_M^{-1} and F are diagonal matrices, this change is simple. Instead of using H^{-1} when computing the search direction, we could use

$$H_F^{-1} = H^{-1} + \begin{pmatrix} 0_N & 0 \\ 0 & F \end{pmatrix},$$

and get the error term $Ap = E(F)q$. This error can easily be made suitably small by choosing F large enough.

However there are other factors that determine the quality of a search direction. For a convex function, using the exact Hessian when computing p gives Newton's method quadratic convergence in the neighborhood of the solution. Although this quadratic convergence is rarely seen in practice with such a non-quadratic function as the logarithmic barrier term, making the change to the Hessian suggested above reduces the rate of convergence considerably. Numerical tests have shown that corrections that are small enough to give an acceptable rate of convergence were not always able to reduce the condition number of $AH^{-1}A^T$ sufficiently.

Iterative Refinement

For a general square matrix B, the error in a solution x of the system $Bx = y$ can often be reduced by performing *iterative refinement*. It involves repeatedly computing the *residual* $r = y - Bx$ and solving $Bz = r$ to give a better solution $x' = x + z$. No additional accuracy can be expected with iterative refinement when the first x was found by Gaussian elimination (of a reasonably well-scaled matrix) and r was computed to the same precision as x (see e.g. [GVL83]).

In our case we do not have the option of calculating an $r = AH^{-1}\bar{g}_L - AH^{-1}A^Tq = Ap$ to more than the precision generally used for all variables. Also, the Cholesky factorization of $AH^{-1}A^T$ is equivalent to Gaussian elimination.

However, if a diagonal correction F is introduced during factorization, the accuracy of q can be improved when residuals are computed from $AH^{-1}A^T$. The iterative refinement is implemented in the following form:

$g_L \leftarrow \bar{g}_L$
$p \leftarrow -H^{-1}g_L$
repeat
  $LL^Tq = Ap$
  update $\pi \leftarrow \pi + q, \quad g_L \leftarrow g_L - A^Tq, \quad p \leftarrow p + H^{-1}Aq$
until $\|Ap\|$ acceptable.

Convergence can be shown (see e.g. [Bj87b]) if

$$\rho(I - (LL^T)^{-1}AH^{-1}A^T) < 1,$$

where $\rho(\cdot)$ denotes the spectral radius. This translates into a bound on the size of F.

Convergence criteria for the residual $r_A = Ap$ are twofold. First, a static upper bound on acceptable values for $\|r_A\|/\|AH^{-1}\bar{g}_L\|$ is given. We choose this conservatively to be $\epsilon_M^{2/3}$. Second, little progress in reducing $\|r_A\|$ is taken as a sign that the remaining residual is inevitable. Average cases show a reduction of $\|r_A\|$ by a factor of about 10^{-2} for every iteration of the refinement.

Applying iterative refinement to the normal equations is equivalent to refining the KKT-system

$$\begin{pmatrix} H & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} -p \\ q \end{pmatrix} = \begin{pmatrix} \bar{g}_L \\ 0 \end{pmatrix},$$

and using normal equations at each step. The residual of this system is

$$r' = \begin{pmatrix} \bar{g}_L + Hp - A^T q \\ Ap \end{pmatrix} = \begin{pmatrix} 0 \\ Ap \end{pmatrix},$$

since $-Hp = -g_L = \bar{g}_L - A^T q$ independently of the accuracy of q. Forming the normal equations for the KKT-system with r' as the right-hand side yields a right-hand side $r_A = Ap$ for the normal equations, as before.

Chapter 7

Inside the Factorization

The first step towards the computation of the Newton search direction is the solution of the normal equations

$$AH^{-1}A^Tq = AH^{-1}g_L.$$

This system is solved by computing the *Cholesky factorization*,

$$AH^{-1}A^T = LL^T$$

and solving the triangular systems

$$Ly = AH^{-1}g_L \qquad \text{and} \qquad L^Tq = y.$$

The time for computing the search direction dominates the time per iteration — typically 80%–90% of the total for a medium-size problem, but it can be as high as 99% for the largest problems. For the linear programs of interest, the matrices A and, to a lesser degree, $AH^{-1}A^T$ and L are sparse, meaning that almost all their elements are zero. An efficient way to form and factorize these large sparse matrices is therefore crucial to this implementation of the barrier method.

Other interior-point methods share this need, since they also solve a symmetric positive-definite systems of the form ADA^T, with D diagonal. (See Adler *et al.* [AKRV87] for programming techniques, or Monma and Morton [MM87]). Thus many of the following observations are equally relevant to these methods.

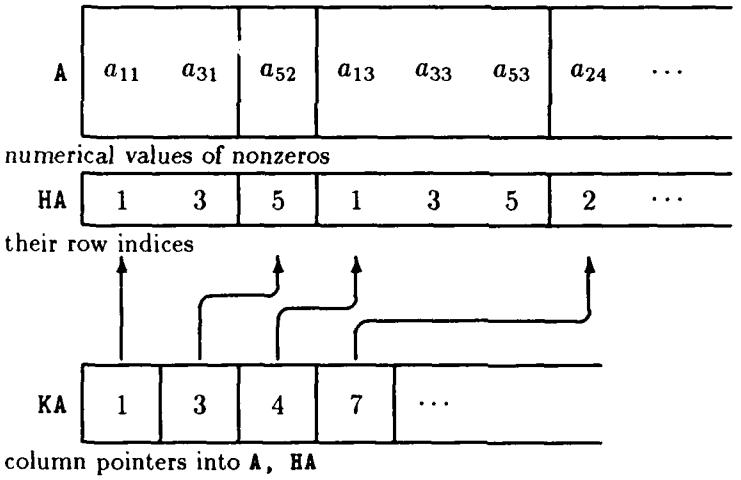
In our implementation the Cholesky factorizations is performed by the subroutines of SPARSPAK-A by Chu, George, Liu and Ng [CGLN84], with minor modifications.

The actual numerical factorization is preceded by an Analyze Phase, in which the nonzero patterns of the involved matrices are analyzed and the necessary data structures are established. These procedures are covered in Chapter 9. For the scope of this chapter we shall ignore the problem of dense columns in A. Extensions to algorithms and data structures taking that issue into account will be described in Chapter 8.

Fundamentals of Sparse Matrix Algebra

In order to avoid storing the large number of zero elements and doing redundant floating-point operations on them, a special data structure is needed for a sparse matrix. It replaces the two-dimensional array used for dense matrices.

Let A be a matrix with n columns, m rows and n_e nonzero elements, where $n_e \ll nm$. The standard approach to store A is to sort its nonzero elements by column into one single array of length n_e (here denoted by **A**). A second array **HA** of the same length records the row numbers of these entries. Each column in this pair of arrays is then found with the help of an array **KA** that contains the position of the first nonzero of that column in **A**. The number of nonzeros in a column j is determined as the difference between to consecutive column offsets in A, here $\mathtt{KA}(j+1) - \mathtt{KA}(j)$. The array **KA** must therefore have one more entry than there are columns in A, with the last value being one more than the length of **A**, i.e., $\mathtt{KA}(n+1) = n_e + 1$. (Clearly an equivalent scheme can be used that sorts the nonzeros of A by row rather than by column.)



The Data Structure for Sparse Matrix Storage

Integer arrays used to access nonzero elements are frequently referred to as *overhead storage*. We assume here that row and column indices fit into two bytes, i.e., $m, n < 2^{15}$, whereas no such assumption is made for the number of nonzeros n_e. Thus, in a FORTRAN implementation **HA** can be an array of short integers and **KA** has to be an array of full integers. With four `INTEGER*2` variables, or two `INTEGER` variables, taking the space of one `DOUBLE PRECISION` word, the primary storage required for A is n_e words, with overhead

storage of $\frac{1}{4}n_e + \frac{1}{2}n$ words.

For the rest of this chapter a *sparse vector* or a *sparse matrix* will be a vector or matrix whose nonzero elements are stored in the described way. A *dense vector* or a *dense matrix* denote a vector or matrix stored in the usual way, regardless of the actual proportion of zero to nonzero elements in them. Assignments between a sparse vector and a dense vector will refer to the copying of nonzero elements of the dense vector to or from a sparse data structure. Row i of a matrix A will be denoted by a_i', and column j by a_j.

Several observations are in order. First, a given element a_{ij} of a sparse matrix cannot be accessed without doing a search along its column j for an entry in HA with value i. For efficiency reasons any sorting and searching of elements should be avoided in these computations, with the exception of the Analyze Phase. The numerical operations we do on sparse matrices should thus be restricted to those that work sequentially on whole columns.

The set of sparse vector operations that do not require sorting, searching or additional workspace include scaling a sparse vector, $s_1 = \alpha\, s_0$; adding a multiple of a sparse vector to a dense vector, $d_1 = d_0 + \alpha\, s$; and computing the inner product of a sparse vector with a dense vector, $\beta = d^T s$, or with itself, $\beta = s^T s$. Not included in this set are operations such as the inner product of two sparse vectors, $\beta = s_1^T s_2$; or their sum, $s_3 = s_1 + s_2$, in the case when the result is to be treated as a sparse vector.

With A stored by its sparse columns a_j, the product $d = Ax$ is computed as $d = \sum x_j a_j$, whereas the product with the transpose $e = A^T y$ is composed of $e_j = a_j^T y$. Here x, y, d and e are assumed to be dense. Observe that the elements of a_j do not have to be sorted by row index in HA for these operations. This fact gives a degree of freedom that we will exploit during the factorization, below.

Forming $AH^{-1}A^T$

Let B denote an $m \times m$ matrix containing the lower-triangular half of $AH^{-1}A^T$,

$$B = \mathrm{tri}(AH^{-1}A^T) \quad \text{where} \quad b_{ij} = \begin{cases} 0 & \text{for } i < j \\ (AH^{-1}A^T)_{ij} & \text{for } i \geq j. \end{cases}$$

Since only half of a symmetric matrix is stored in practice, the matrix to be formed is actually B.

One way to compute this scaled outer product is by computing each element b_{ik} as a scaled inner product of rows of A,

for $k = 1 \ldots m$

  for $i = 1 \ldots k$

    $b_{ik} = a_i' H^{-1} a_k'^T$.

This is not easily implemented for a sparse A, because even if A is stored by rows rather than by columns, the inner product of two sparse vectors is not an efficient operation. Computing B by explicitly adding the column outer products $B = \sum 1/h_{jj}\ \mathrm{tri}(a_j a_j^T)$ is not possible without keeping B temporarily in a dense representation, which requires prohibitively much memory.

One solution is a scheme that rearranges the second loop of the algorithm above and requires a dense vector d as temporary storage. Let a_j^k define the lower part of a_j, so that

$$(a_j^k)_\ell = \begin{cases} 0 & \text{for } \ell < k \\ a_{\ell j} & \text{for } \ell \geq k\,. \end{cases}$$

The algorithm for forming $AH^{-1}A^T$ then becomes

for $k = 1 \ldots m$

  $d = 0$

  for j such that $a_{kj} \neq 0$

    $d = d + (a_{kj}/h_{jj})\, a_j^k$

  $b_k = d$.

Here only columns of B and A are accessed. The question of finding the elements of a_j^k without searching for them in the column a_j will be addressed on page 45.

Factorizing $AH^{-1}A^T$

Since $AH^{-1}A^T$ is a symmetric positive-definite matrix, there exists a Cholesky factorization $AH^{-1}A^T = LL^T$, where L is lower triangular. The method to compute it should have the property that B gets overwritten by (part of) L. Of the three methods with this property described by George and Liu [GL81], the *inner-product form* is used in the SPARSPAK package:

for $k = 1 \ldots m$

$l_{kk} = b_{kk} - \sum_{j=1}^{k-1} l_{kj}^2$

for $i = k+1 \ldots m$

$l_{ik} = b_{ik} - \sum_{j=1}^{k-1} l_{kj} l_{ij}$

$l_k = (1/\sqrt{l_{kk}})\, l_k$.

As before, the inner product of two rows is avoided by reformulating the second loop, using l_j^k as the lower part of l_j :

for $k = 1 \ldots m$

$d = b_k$

for j such that $l_{kj} \neq 0$

$d = d - l_{kj}\, l_j^k$

$l_k = (1/\sqrt{d_k})\, d.$

When examining this procedure for the resulting nonzero structure of L, note that L has a nonzero wherever B has one, but might have more. This property allows a general B to be stored in the same sparse matrix structure as L. (This feature becomes irrelevant for the special case of $AH^{-1}A^T$ by the observation in the next section.)

The minor effort of computing m square roots can be saved by using the factorization $AH^{-1}A^T = LDL^T$ instead. Here L is an unit lower triangular matrix and D is diagonal (see [AKRV87]). Since taking this approach would add several scalings with D^{-1} to the computation of the Schur complement in Chapter 8, its advantages in our implementation are not clear and this path was not taken.

Forming and Factorizing in one Step

The similarities between the two algorithms sketched above are striking. Both arise from an inner-product form; both have a dense vector to accumulate multiples of lower parts of sparse columns; in both cases this depends on the column element in the current row.

The outer loops of both operations have an index running over the same range, one ending with $b_k = d$, the other starting with $d = b_k$. Checking that b_k is not accessed when computing any l_j with $j < k$, we see that it is not necessary to form B explicitly. Instead we can form and factorize each column of L in one step.

for $k = 1 \ldots m$

$d = 0$

for j such that $a_{kj} \neq 0$

$d = d + (a_{kj}/h_{jj})\, a_j^k$

for j such that $l_{kj} \neq 0$

$d = d - l_{kj}\, l_j^k$

$l_k = (1/\sqrt{d_k})\, d.$

(The last line stands for $l_{ik} \leftarrow d_i/\sqrt{d_k}$ whenever l_{ik} is a nonzero of L. Since these indices i are known in advance, it suffices to reset the corresponding d_i to zero, instead of zeroing out the whole vector.)

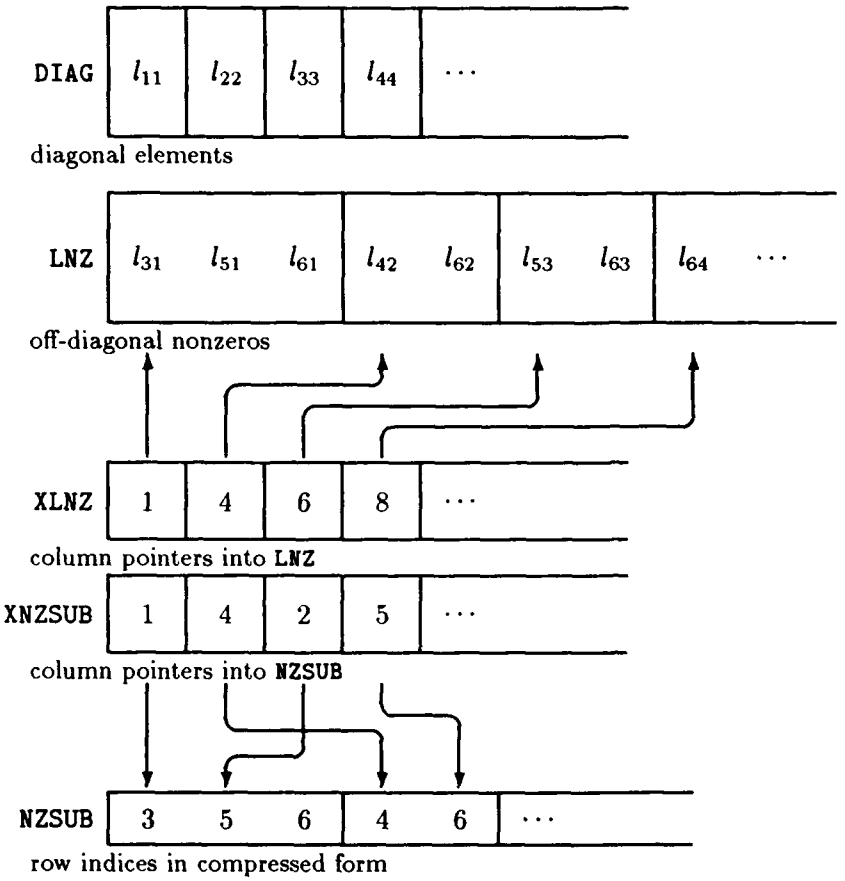
Data Structure for L

The storage scheme for L deviates a little from the standard way of storing sparse matrices in order to take advantage of two properties of this matrix.

- We assume A has no zero rows, so that all diagonal elements of L are nonzero. This diagonal can be stored separately and the column-wise sparse storage is only applied to the off-diagonal elements.
- The distribution of nonzeros in L has a special form. A column l_k contains nonzeros in all rows where there are nonzeros in l_j^k for j with $l_{kj} \neq 0$. This leads to the common occurrence of patterns of row indices that repeat themselves for different columns. To save some overhead storage, SPARSPAK uses a *compressed* scheme where repeated patterns are stored only once in the array **NZSUB** and a second array of pointers **XNZSUB** points to the sequence of row indices for each column. **XNZSUB** does not have the property that the difference between two adjacent entries is the number of nonzeros in one column. However, this number can still be derived from the first array of pointers **XLNZ**.

Dynamic Pointers

The algorithm as explained so far leaves two questions unanswered. (1) How do we find the columns j that affect the formation of column k, namely those with $a_{kj} \neq 0$ or $l_{kj} \neq 0$, respectively? (2) How do we access the part needed, a_j^k or l_j^k, when individual elements cannot be addressed without some searching? Since these issues apply equally to the forming and the factorizing step, they will be discussed using the notation involved



Data Structure for the Factor L

in forming $AH^{-1}A^T$, while the corresponding notation for factorizing $AH^{-1}A^T$ will be mentioned in parentheses.

The columns accessed when forming column b_k (l_k) are those sharing nonzero elements with row a'_k (l'_k). There are two methods for finding the nonzeros of a row, one static and one dynamic.

The static method employs an equivalent data structure to KA/HA for the rows. An array JA of length n_e records the column indices of all nonzeros of A sorted by row, and an array KArow stores the pointers into JA for each row. Both can be constructed during the Analyze Phase.

The dynamic alternative uses linked lists. Here a *link* is associated with every column and a *list header* with every row. At the time column k is computed, the list for row k contains indices of all columns that have a nonzero in that row. Afterwards each column j of this set is linked to the row i that contains its next nonzero, i.e., $i = \min\{i > k \mid a_{ij} \neq 0\}$.

The next nonzero in a column can be found without searching when the nonzeros of each column are sorted by row index inside A/HA. So, unlike simple matrix operations, such a scheme requires a special order for A. (See also 'Permutations', below.)

The ordering is also needed to answer the second question, the problem of accessing a_j^k (l_j^k). A second, dynamic array of pointers over all columns KA1ST (FIRST), initially set to KA (XLNZ), is used to point to the first element of a_j^k (l_j^k) for the next column k that is going to use column j. Again, this array can be updated after column k is computed simply by adding one, so that, say KA1ST(j) now points to the first element of a_j^i, where i is the row with the next nonzero, as above. The last element of any a_j^k is the same as the last element of a_j and sits at offset KA($j+1$) $-$ 1. This way of accessing the lower part of the column is independent of whether the column was found by the static or dynamic method above.

The similarity of the algorithms for forming and factorizing should imply that we choose the same data structure for finding the right column in both cases. However, the determining dimensions are not of the same order. Additional overhead storage needed for the alternatives are (in terms of full INTEGERs)

	Forming	Factorizing
static	$n_e(A)/2 + n$	$n_e(L)/2 + m$
dynamic	$(n+m)/2$	$m/2$ [5]

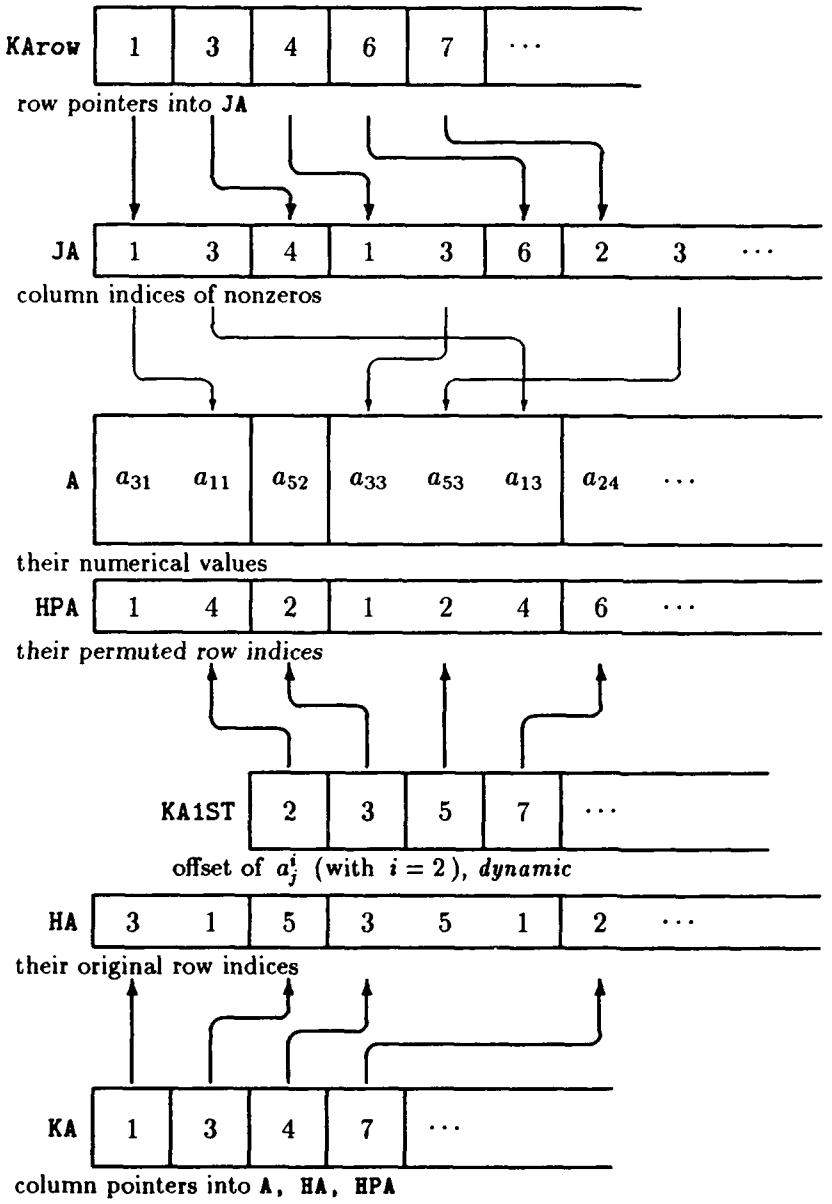
As we expect there to be more off-diagonal nonzeros in L than nonzeros in A, the time savings from avoiding the maintenance of a linked list in the static method are gained at the expense of more memory during the factorization. In our implementation we leave intact the linked list used in SPARSPAK for L, but use the static access scheme for the rows in A.

Permutations

The outline of the procedure to factorize $AH^{-1}A^T$ has omitted one important aspect. Although the number of nonzeros in $AH^{-1}A^T$ is independent of the row permutation of A, the number of nonzeros in L is not. During the Analyze Phase, see Chapter 9, a permutation P is sought that minimizes the fill-in of L. Thus the actual factorization is of the form

$$PAH^{-1}A^TP^T = LL^T.$$

[5] For the factorization, both the list headers and the links can use the same array of length m since the off-diagonal nonzeros of row i can only lie in columns j with $j < i$.



Data Structure for A during factorization

By rewriting $PAH^{-1}A^TP^T$ as $(PA)H^{-1}(PA)^T$, we leave the factorization as it is, just using PA instead of A. This is done without any additional work by introducing an additional array `HPA` that contains the permuted row index for each nonzero. Sorting the entries of `A`, `HA` and `HPA` so that the entries in `HPA` are in ascending order for each column, completes the adjustments that have to be made for the permutation. All this is done during the Analyze Phase.

Gay [Gay88] suggests that some permuting of vectors can be saved when columns of L are stored in the order of the corresponding row indices of A. This idea was not followed here, since keeping L and other vectors and matrices in the permuted order is essential for some details of the implementation of the Schur complement (see page 55).

Chapter 8

When Things get too Dense

The sparse constraint matrix of the linear program in Phase I can be written in the form $A = (A_N \; a \; I)$, where a denotes the artificial column. Since a is assumed to have no structural zeros, aa^T and thus $AH^{-1}A^T$ are dense matrices. A similar problem can also occur in the optimality phase, i.e. when $a = 0$. Including all the columns of A_N when forming $AH^{-1}A^T$ can sometimes be uneconomical at best, often making it impossible to run a certain LP on a memory-constrained machine. The columns we would like to omit from $AH^{-1}A^T$ will be called *dense columns*, although factors other than the mere number of nonzero elements might contribute to the definition of this set, see page 52.

Schur Complement

Assume that there are n_d dense columns (including a), and let A_d be the submatrix of A_N that contains them. Defining the partitions

$$A = \begin{pmatrix} A_s & A_d & I \end{pmatrix} \qquad \text{and} \qquad H = \text{diag}(H_s, \; H_d, \; H_M)$$

we have

$$AH^{-1}A^T = A_s H_s^{-1} A_s^T + H_M^{-1} \quad + \quad A_d H_d^{-1} A_d^T,$$

where $A_s H_s^{-1} A_s^T + H_M^{-1}$ has a sparse triangular factor L.

Taking the matrix square root $V = A_d H_d^{-1/2}$ of the remainder, the solution of the normal equations $AH^{-1}A^T q = y$ can be found from the larger system

$$\begin{pmatrix} LL^T & V \\ V^T & -I \end{pmatrix} \begin{pmatrix} q \\ z \end{pmatrix} = \begin{pmatrix} y \\ 0 \end{pmatrix}.$$

(Equivalent formulations, with $V = A_d$ or $V = A_d H_d^{-1}$ and H_d or H_d^{-1} in the lower right-hand corner, yield somewhat cleaner notation but proved to be less stable numerically.)

The matrix $C = I + W^T W$, where $W = L^{-1}V$, is called the *Schur complement* of LL^T. The required solution q may be determined by solving the following sequence of equations:

$$\begin{aligned} Ls &= y \\ Cz &= W^T s \\ L^T q &= s - Wz. \end{aligned}$$

The numerical accuracy of the solution may be improved by iterative refinement, see page 38, when $n_d > 0$. In practice we have observed that only one additional refinement step is worthwhile. In the rare case where the residual $r_A = y - AH^{-1}A^T q$ turns out to be already very small, this step may be skipped.

Comparison with preconditioned CG

Dense columns of A present a difficulty for all interior-point methods that factorize a matrix of the form ADA^T with D diagonal. An alternative approach is to use the Cholesky factor L of $A_s H_s^{-1} A_s^T + H_M^{-1}$, as a preconditioner for a conjugate-gradient method (see page 32).

The two approaches are not the only alternatives. A hybrid method is possible, where the Schur-complement method works inside CG. The preconditioner could be improved this way to include dense columns a_j where h_j^{-1} is large. Or the conjugate-gradient method can be employed to cope with corrections made to L during the factorization. Such a hybrid method was not tested in the scope of this research.

We shall compare the Schur-complement method to a CG implementation that uses `LSQR` [PS82]. To have some measure of the work performed beyond the factorization, we identify two important parts, namely the *solves*, $L^T x = b$ or $Lx = b$, and the *products*, Au or $A^T x$, for some vectors x, u. CG needs 2 solves and 2 products for start-up in addition to 2 solves and 2 products per iteration. Whereas we can treat the Schur-complement method with two steps of iterative refinement as a direct method, `LSQR` is an iterative method. Typically it was observed to take 2 iterations in the case of one dense column and some i iterations in the case of $n_d > 1$ dense columns, with $n_d + 1 \leq i < 2\,n_d$.

To compare the work per barrier iteration with the two approaches three cases are considered:

- $n_d = 0$. In this case $AH^{-1}A^T = LL^T$ and the normal equations can be solved directly. Neither CG nor the Schur complement need be employed, both are identical here.

• $n_d = 1$. One solve is needed to compute W, plus two per step of the iterative refinement. Computing the residual costs two products. Any additional work is negligible. This adds up to 5 solves and 2 products, which compares favorably to 6 solves and 6 products with CG. Typically the dense column here is a, which has a density of 100%. This distinguishes it from the next case:

• $n_d > 1$. Again two solves are needed per step of the iterative refinement and two products to form the residual. Computing W takes exactly n_d solves, giving a total of $n_d + 4$ solves and 2 products. CG takes at least $2n_d + 4$ solves and $2n_d + 4$ products. The additional work needed to compute the Schur complement $C = I + W^T W$ is significant here, but so are the savings obtained by taking advantage of the sparsity in V when forming $L^T W = V$.

Comparison of Storage required

Here the case $n_d = 0$ is not relevant, since storage always must be allocated for the maximum need, which is during Phase I. Storage requirements for the Schur-complement are less in the case $n_d = 1$, since some work vectors needed by LSQR can be saved in an efficient implementation.

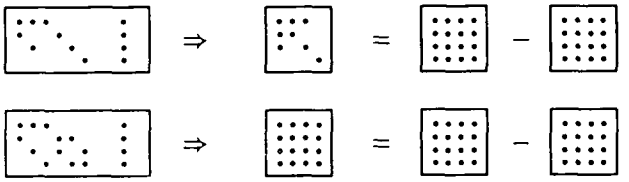
Analytically it is unclear, however, how the methods compare in the case $n_d > 1$. Although W must be stored, this can be done effectively in some sparse format (page 55), since it is typically only 25%–75% dense. Because of the great time advantage, the time-optimal choice for the number of dense columns is bigger than with CG. This in turn reduces the density of L considerably. Experiments with a small number of LP test problems with dense columns, suggest that the size of L and W together for a time-optimal choice of A_d is substantially less than the size of L alone for a choice of A_d that would be optimal with a conjugate-gradient method.

Identifying a Dense Column

The definition used in our implementation is simple: if a column has more than a preassigned number of nonzeros, it is handled as a dense column. This threshold number can be set by the user. If it is not specified it defaults to a rule of thumb involving the number of rows m. [6]

[6] The number used is set up to make a near-optimal choice for the four or five problems of the test set with dense columns: $\sqrt{3m + 700}$.

For a general-purpose implementation, a better way of identifying dense columns may be required. The underlying assumption in our definition is that the positive effect on the efficiency of the factorization achieved by taking out column a_j, is increasing in the number of nonzeros $n_z(a_j)$. This assumption does not necessarily hold. The effect depends quite heavily on the nonzero structure of the other columns. Taking out the column with the most nonzeros might sometimes have less effect on $n_z(AH^{-1}A^T)$ than taking out a column with relatively few nonzeros. The effect on $n_z(L)$ is even harder to predict.



$$A = \begin{pmatrix} A_1 & a \end{pmatrix} \quad \Rightarrow \quad A_1 H_1^{-1} A_1^T = AH^{-1}A^T - aa^T/h_a$$

Taking out a dense column of A does not necessarily improve the sparsity of $AH^{-1}A^T$.

Experiments varying the threshold on a test set limited to the problems with dense columns showed that a time-optimal choice of A_d was also close to storage-optimal (and vice-versa). A heuristic explanation is that the storage consists mostly of the nonzero elements of L and W and the number of operations is in part an increasing function of the number of these. This result implies that gains in speed may be possible by finding the storage-optimal partition A_s / A_d when the storage is allocated during the Analyze Phase. However, finding such a partition involves solving a very hard combinatorial problem that was not tackled in this research. Advances in this direction could show improvements even for problems that are currently not considered to have dense columns.

The problem is even more complex when numerical issues are taken into account. The matrix $A_s H_s^{-1} A_s^T + H_M^{-1}$ is more likely to be nearly rank-deficient than $AH^{-1}A^T$. This implies that measures against ill-conditioning, like the freeing of fixed slack variables (page 34) or adding a diagonal matrix (page 36), will be necessary more often. These measures may require more steps of iterative refinement or even result in more minor iterations.

Going to the Extreme

Of interest is the extreme case in which all columns are treated as dense, i.e., $A_d = A_N$. The matrices of the Schur-complement method for this case can be given as $L = H_M^{-1/2}$, $V = A_N H_N^{-1/2}$, $W = H_M^{1/2} A_N H_N^{-1/2}$ and

$$C = I + H_N^{-1/2} A_N^T H_M A_N H_N^{-1/2}.$$

Usually a Schur complement C of that form can no longer be efficiently treated as a dense matrix. It has to be stored in a sparse form and factorized accordingly. By ignoring all the diagonal matrices in the formula for C, its sparsity structure can be identified to be that of $A_N^T A_N$. The computational effort for this method is, therefore, about the same as that for the null-space method of page 33.

As previously mentioned during the discussion of the null-space method, algorithms based on factorizing a matrix of the form $A_N^T A_N$ are likely to be less efficient than algorithms based on factorizing a matrix of the form $A_N A_N^T$. There are, however, implications of this extreme case for the way we look at the Schur-complement method. Choosing some partition $A_s \,/\, A_d$ can be viewed as striking a compromise between the nonzero structures of $A_N^T A_N$ and $A_N A_N^T$ — a compromise with the promise of being more efficient than both extremes.

Implementation details

The procedure for solving the system $L^T x = b$ usually involves two systems of equations, $L^T x_p = Pb$ and $Px = x_p$, where P is a permutation matrix. The permutation is the minimum-degree ordering found during the Analyze Phase, see Chapter 9. Some economy of speed (and storage, see below) can be achieved by keeping the intermediate vectors and matrices in the permuted order. In the following, a subscript p denotes a vector or matrix permuted by P.

In detail, the actual sequence of computations is

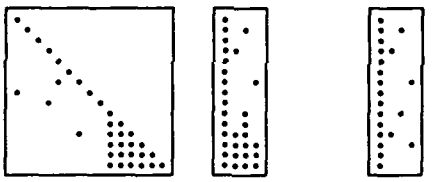
$$\begin{aligned} LW_p &= PV \\ C &= I + {W_p}^T W_p \\ Ls_p &= Py \\ Cz &= {W_p}^T s_p \\ L^T q_p &= s_p - W_p z \\ Pq &= q_p. \end{aligned}$$

Since PA is implemented in the form of a second row index vector for A (see page 47), accessing elements of PA (or PV) involves the same work as accessing elements of A.

The Schur complement C is stored and factorized as a dense triangular matrix. Since n_d is usually very small, solving for z takes a negligible amount of time.

Cluster Storage

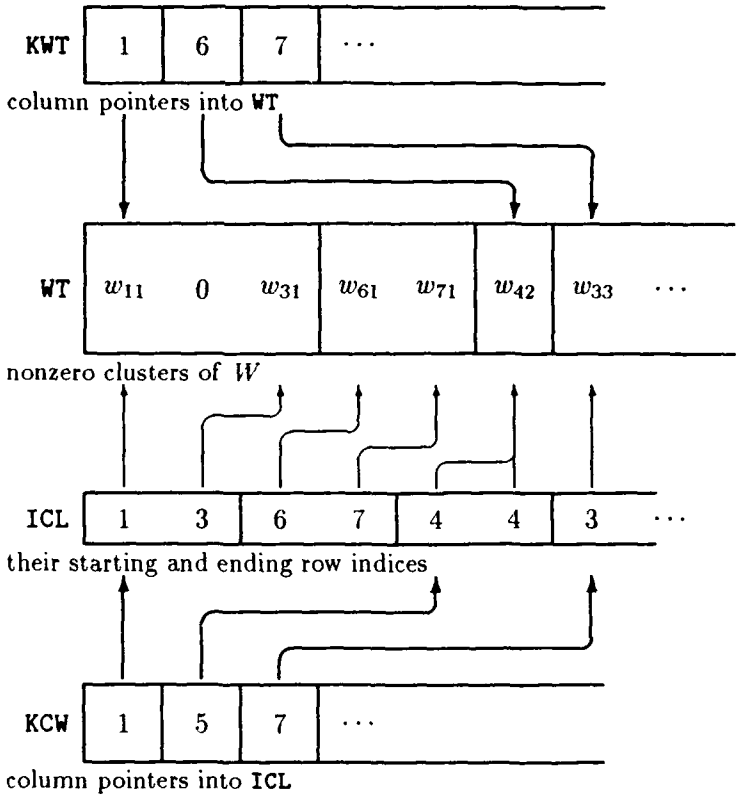
The data structure for W is special, since this matrix is medium dense. Storing it in conventional sparse form would add considerable overhead to the computation, especially in the case $A_d = a$, where W is all dense. But other columns of W are expected to have a density in or beyond the 25%–50% range also, where dense storage schemes become more effective on most scalar processors. Since W is the result of a triangular solve and is stored permuted as W_p, most of the nonzeros are clustered towards the lower end of the columns. This is especially so because the minimum-degree ordering tends to give a dense (triangular) submatrix in the lower right-hand corner of L.



$L \qquad W_p \quad = \quad V_p$

In order to have one data structure that is effective for both dense and somewhat sparser columns, the scheme we have used indexes *clusters of nonzeros* instead of the nonzeros themselves. This reduces the integer overhead by about one third compared to real sparse storage, while retaining some of the computational advantages of dense vector handling.

In detail, a cluster is defined as a sequence of consecutive nonzeros in one column. Consecutive is meant here in terms of the minimum-degree ordering of rows in which W_p is stored. Additional time savings can be obtained if we allow a small number of zeros to be included in the cluster. (Only for single zeros did this seem to be worthwhile on our machines.) An indexing array `ICL` is maintained, storing the first and last row index of each cluster. An outer index array `KCW` points to the entry of `ICL` belonging to the first cluster of each column. The `DOUBLE PRECISION` array `WT` contains all the clusters (including those single zeros) and a second outer index array `KWT` points to the first nonzero of each column.



Cluster storage for W

If there are n_w nonzeros in W and n_c clusters can be found, the total storage in bytes for this scheme is little more than $6m + 4n_c + 8n_w$. Dense storage would take $8n_d m$ and sparse storage $4m + 10n_w$ bytes. The advantage for the cluster form therefore disappears when the average cluster length falls below 2. Roughly the same trade-off may be expected for accessing speeds.

Although operations on clusters are dense by nature, there is a considerable disadvantage in calling standard subroutines to handle simple vector arithmetic for them, as the clusters tend to be rather short (rarely more than 10 elements).

Chapter 9

The Analyze Phase

The Cholesky factorization of a sparse symmetric positive-definite matrix is a well-studied problem; see, e.g. [GL81] and [DER86]. The factorization is generally done in two phases, the *Analyze Phase* and the *Numerical Phase*. In the Analyze Phase, the structure of the nonzero elements in the matrix is analyzed and a suitable data structure and order of operations is established. In the Numerical Phase these operations are then executed. For typical matrices each phase takes about an equal amount of computer time. It is important to note that the numerical values of the matrix elements are not relevant in the Analyze Phase. This is true because for positive-definite matrices all orderings are acceptable as far as numerical stability is concerned.

The systems of equations solved at each iteration of an interior-point methods are a special application of these factorization techniques. The matrices $AH^{-1}A^T$ have the same nonzero structure for every iteration, independent of the values of H^{-1}. This leads to a method that need only have one Analyze Phase and several Numerical Phases. We shall therefore refer to the Analyze Phase of the barrier algorithm, which performs all non-numerical setup steps in preparation for forming and factorizing $AH^{-1}A^T$.

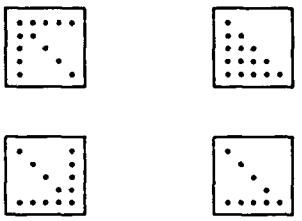
Steps of the Analyze Phase include: (1) most importantly, the search for an ordering of the matrix $AH^{-1}A^T$ that reduces the fill-in in its factor L; (2) the sorting of A according to this ordering; and (3) the generation of data structures for the Schur-complement procedure to handle dense columns.

The Minimum-Degree Ordering

It is a characteristic of both symmetric and unsymmetric systems that the ordering of rows and columns has a great deal of influence on the number of nonzeros in the factors, although it does not change the number of nonzeros in the original matrix. Despite the existence of

some counterexamples, the factorization time is generally observed to be increasing in the number of nonzeros in the factor.

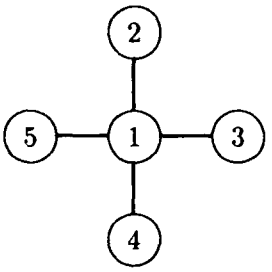
To see the impact of the ordering of the matrix on the nonzeros in the factor, consider the following simple example. Take a symmetric matrix that is zero except for its diagonal and the first row/column. Its Cholesky factor will be a dense triangle. If the first row and column are interchanged with the last, however, only the diagonal and the last row are nonzero in the resulting factor.



matrix factor

The second matrix is said to suffer no *fill-in* during the factorization. This expression reflects the fact that for every nonzero in the lower triangular half of the matrix there will be a nonzero in the factor. Any additional nonzeros in the factor are considered as *filling* the blank space in the matrix.

The problem of finding the row and column ordering that minimizes the number of nonzeros in the factor is NP-complete (see [Yan81]). Efficient heuristics have been discovered that give a near-optimal ordering. The most prominent is the *minimum-degree ordering.* (See [GL87] for recent improvements.) Its name is derived from the graph-theoretic representation of the problem that will be sketched here.



The sparsity pattern of a symmetric matrix can be represented by an undirected graph. The graph has a node for every row/column of the matrix, and an edge from node i to

node j wherever there is a nonzero at (i,j). The graph representation of our example above is thus a star with node 1 at its center.

The graph equivalent to computing column j of the factor is removing node j from the graph and adding an edge (i,k) for every pair of nodes i and k that were previously connected to node j by edges (i,j) and (j,k). The new edge (i,k) corresponds to fill-in in the factor, if the nodes i and k have not been connected before.

In order to minimize fill-in, i.e., to minimize the number of edges added, the node with the minimum number of outgoing edges is removed first. Since the number of adjacent edges is also called the *degree* of a node, this rule constitutes the minimum-degree algorithm. Variants of the algorithm differ in the way ties are resolved, and in the frequency with which the degree is recomputed.

In our little example the minimum-degree algorithm will yield an ordering where node 1 is either last or second to last. All such orderings are optimal. This is due to the fact that the original graph was a tree, but in general we would not expect the minimum-degree ordering to generate the least fill-in possible.

The reason for the popularity of the minimum-degree algorithm lies both in the quality of the resulting ordering and in the efficiency of some of its implementations. (The version used in our implementation is SPARSPAK's `GENMMD` routine, using a "multiple minimum external degree" method.) This is to imply that the minimum-degree algorithm was found to be the best trade-off between the time invested in the Analyze Phase and the time saved during the Numerical Phase. However, most research in this area assumes that only one Numerical Phase is performed per Analyze Phase. For the case of interior-point methods, there is some potential for more expensive ordering methods in the Analyze Phase, since their cost is amortized over a greater number of factorizations. Adler *et al.* [AKRV87] report some success with a *minimum local fill-in* method.

Cliques

The normal input format for the ordering algorithm is a list of the row and column indices for each nonzero in the matrix. The first step of the minimum-degree algorithm is to convert that list into a data structure that represents the adjacencies of the graph.

In order to generate such a list for $AH^{-1}A^T$, the locations of nonzeros have to be determined by forming the symbolic product AA^T. The product is symbolic in the sense that there is no real arithmetic involved, but the nonzero patterns of rows are compared.

Forming this symbolic product can be avoided by observing that the graph of AA^T consists of *cliques*. A clique is a set of nodes where each pair of nodes is connected by an edge. The graph representing the outer product $a_j a_j^T$ is a clique. This clique consists of the nodes that correspond to the rows of the nonzeros in a_j. Since AA^T is the sum of outer products $a_j a_j^T$, its graph representation is the union of the corresponding cliques.

Using a special SPARSPAK input routine, the nonzero structure of AA^T is simply represented by the series of cliques corresponding to the columns of A. Dense columns and columns corresponding to fixed variables are ignored for this purpose.

Sorting A

Once the row ordering is determined, the formation of $AH^{-1}A^T$ in the Numerical Phase can be made considerably more efficient by sorting A. This is done by reordering the entries of the arrays `A` and `HA` (refer to Chapter 7) so that the nonzeros in each column are in that permuted order. Since dense columns are not included in the Cholesky factorization, they do not need to be sorted. Consequently the sorting algorithm does not have to be sophisticated, because the number of nonzeros to sort per column is small. The time spent on sorting is almost negligible.

Memory allocation and Data Structures

The memory requirements of the minimum-degree algorithm are considerable and are not bounded by a reasonable function of the dimensions of A. In some instances the memory required in the Analyze Phase can exceed that of the Numerical Phase.

All memory left after loading `A` and allocating space for its overhead storage and the main vectors is first reserved for the minimum-degree algorithm. After the ordering is found, the memory is reassigned to the data structure that represents L and its overhead.

Given the ordering, the array `HPA` of permuted row indices is generated (page 47). The arrays `JA` and `KArow` that allow row-wise access to the nonzeros of A are determined by searching through all non-dense columns.

With these data structures in place, a symbolic solve $LW = A_d$ is performed to determine the nonzero structure of W. The columns of W are then searched for clusters and the corresponding integer arrays are generated.

All additional memory is temporary workspace, i.e., contains arrays that are recomputed at each iteration. These include the pointers needed during the factorization, the numerical values of W and C, and some work vectors.

Part III Testing

Chapter 10

Test Set and Setting

An implementation was developed to test the various facets of the algorithm discussed in Parts I and II. Since randomly generated problems prove to give poor insight into the behavior of a large-scale algorithm, a collection of real-world problems was used as the test set.

The relative performance of different algorithms always depends somewhat on the hardware and software used for the test. With linear programming, these dependencies seem to be more critical for interior-point methods than for the simplex method.

In addition to presenting the general results of performance tests, we discuss in this chapter the influence that the test set and the computing environment may have on the performance.

The Test Problems

The set of test problems consists of the first 53 problems in the *netlib* collection [Gay85]. They are available via electronic mail and have come to be regarded as a standard benchmark for linear programming algorithms. In the following tables, the problems are ordered according to the number of nonzero elements in A as in [Lus87].

Although 53 problems is a reasonably large set, many of the problems are related. The performance of related problems is often correlated. No claim is made that the problems are typical of LP problems commonly solved. Indeed that the problems ended up in a test set may be some indication the problems are atypical, i.e., hard to solve by the simplex method. Too much, therefore, should not be concluded from the results. Batch testing of this type is perhaps best viewed as a means for detecting bad algorithms.

Apart from the use of row and column scaling where indicated, each problem was solved as given. No attempt was made to simplify the problems by first preprocessing them,

LP name	rows	columns	nonzeros	% of nonz.	fixed rows	upper bounds
AFIRO	28	32	88	9.8	8	0
ADLITTLE	57	97	465	8.4	15	0
SC205	206	203	552	1.3	91	0
SCAGR7	130	140	553	3.0	84	0
SHARE2B	97	79	730	9.5	13	0
RECIPE	92	180	752	4.5	67	69
VTPBASE	199	203	914	2.3	55	97
SHARE1B	118	225	1182	4.4	89	0
BORE3D	234	315	1525	2.1	214	12
SCORPION	389	358	1744	1.2	280	0
CAPRI	272	353	1786	1.9	142	131
SCAGR25	472	500	2029	0.9	300	0
SCTAP1	301	480	2052	1.4	120	0
BRANDY	221	249	2150	3.9	166	0
ISRAEL	175	142	2358	9.5	0	0
ETAMACRO	401	688	2489	0.9	272	180
SCFXM1	331	457	2612	1.7	187	0
GROW7	141	301	2633	6.2	140	280
BANDM	306	472	2659	1.8	305	0
E226	224	282	2767	4.9	33	0
STANDATA	360	1075	3038	0.8	160	104
SCSD1	78	760	3148	5.3	77	0
GFRDPNC	617	1092	3467	0.5	548	258
BEACONFD	174	262	3476	7.6	140	0
STAIR	357	467	3857	2.3	209	6
SCRS8	491	1169	4029	0.7	384	0
SEBA	516	1028	4874	0.9	507	507
SHELL	537	1775	4900	0.5	534	126
PILOT4	411	1000	5145	1.2	287	247
SCFXM2	661	914	5229	0.9	374	0
SCSD6	148	1350	5666	2.8	147	0
GROW15	301	645	5665	2.9	300	600
SHIP04S	403	1458	5810	1.0	354	0
FFFFF800	525	854	6235	1.4	350	0
GANGES	1310	1681	7021	0.3	1284	404
SCFXM3	991	1371	7846	0.6	561	0
SCTAP2	1091	1880	8124	0.4	470	0
GROW22	441	946	8318	2.0	440	880
SHIP04L	403	2118	8450	1.0	354	0
PILOTWE	723	2789	9218	0.5	583	296
SIERRA	1228	2036	9338	0.4	528	2016
SHIP08S	779	2387	9501	0.5	698	0
SCTAP3	1481	2480	10734	0.3	620	0
SHIP12S	1152	2763	10941	0.3	1045	0
25FV47	822	1571	11127	0.9	516	0
SCSD8	398	2750	11334	1.0	397	0
NESM	663	2923	13988	0.7	480	1739
CZPROB	930	3523	14173	0.4	890	0
PILOTJA	941	1988	14706	0.8	661	339
SHIP08L	779	4283	17085	0.5	698	0
SHIP12L	1152	5427	21597	0.3	1045	0
80BAU3B	2263	9799	29063	0.1	0	3057
PILOTS	1442	3652	43220	0.8	233	1129

netlib Test Problems

e.g., by eliminating redundant constraints. The experiments were intended to investigate algorithmic performance in precisely the kind of circumstances that such procedures are designed to eliminate. Preprocessing may prove useful within a practical code. ([AKRV87] discusses experiences with it.) However, preprocessing cannot be assumed to eliminate undesirable features of linear programs. This is particularly true for very large problems.

The Computing Environment

All runs were obtained as batch jobs on a DEC VAXstation II. The operating system was VAX/VMS version 4.5. The compiler was VAX FORTRAN version 4.6 with default options, including code optimization and D_floating arithmetic (relative precision $\epsilon_M \approx 2.8 \times 10^{-17}$). Solution times are given in CPU seconds; they do not include time for data input or solution output.

The simplex implementation used for comparison purposes is the Fortran code `MINOS` 5.3 (May 1988). Default values of the parameters were used throughout (see [GMSW88]); these include scaling (`SCALE OPTION 2`) and partial pricing (`PARTIAL PRICE 10`). See also Lustig [Lus87] for a comparison of different parameter settings with `MINOS`.

Memory Constraints

Our implementation of the primal barrier method was designed to keep paging to a minimum within the available memory. This is relevant for two reasons. First, the implementation tests the behavior of an interior-point method in a workstation environment, which recently has evolved as the computer of choice for many linear programming applications. Second, it makes comparisons with the simplex method more meaningful, since `MINOS` works comfortably within this memory constraint.

For the largest problem, `PILOTS`, our implementation requires about 3 megabytes of in-core memory. That includes one copy of A and L, as well as the necessary vectors and integer data structures. `MINOS` requires a little over 2 megabytes to solve `PILOTS`.

On other machines, there are several opportunities to enhance the speed of the factorization by using more memory. Instead of forming $AH^{-1}A^T$ and overwriting it by L at every iteration, it is possible to keep and update $AH^{-1}A^T$ by adding some $A\,\Delta H^{-1}A^T$. By using an approximate H^{-1}, the diagonal of ΔH^{-1} may contain many zeros and make the update $A\,\Delta H^{-1}A^T$ very sparse and efficient to compute, see [AKRV87]. Another option is

to compute and store every product $a_{ij}a_{kj}$ of nonzeros of a column of A once. Forming the elements of $AH^{-1}A^T$ is then accomplished by multiplying these products with h_j^{-1} and adding them up; see [MM87].

Still more memory intensive is the *interpretative procedure*. At the innermost loop of the Cholesky factorization are operations of the type

$$l_{ik} = l_{ik} - l_{kj}l_{ij}.$$

Since all (i,j,k) combinations for which this operation is nontrivial are known during the Analyze Phase, the memory locations involved in each of these operations can be recorded in one very long array. This eliminates a large part of the overhead needed to access the sparse data structures. Adler *et al.* [AKRV87] use this method in one part of the factorization, while treating the other part of L as dense to save memory. Such an approach seems especially promising for machines with vector-type architecture. However, Gay [Gay88] reports that the interpretative procedure rarely saves more than 20% of the factorization time on the *netlib* test set.

In the short history of research on interior-point methods, several implementations were developed that exploit the resources of advanced computers to an extent not common in portable simplex codes. As work on both types of linear programming algorithm continues, it will be interesting to see whether the ability to make use of such resources will give interior-point methods an advantage. This research, however, tries to compare the two using about the same amount of memory and using similar data structures for both.

The Runs

As with the implementation of any other optimization method, many preassigned parameters must be selected. We define a *run* to be a suite of results for a group of test problems that were all solved using the same set of parameters.

The table on page 65 reports on a run that includes all 53 problems. The main characteristics are that the problems are solved unscaled, small bounds are added to fixed slack variables, and the composite objective function uses a fairly small weight $\omega = 10^{-4}$.

The results are reported in terms of the number of (minor) iterations, the optimal value of the linear objective function, the norm of infeasibilities in relation to the norm of the solution x, and the solution time. The last two columns give the corresponding solution time

	Itn.	Obj. fct.	$\|Ax-b\|/\|x\|$	CPU sec.	MINOS 5.3	
AFIRO	20	-4.647529E+02	8.8E-16	2.84	0.49	−−
ADLITTLE	31	2.254951E+05	7.2E-10	10.05	5.07	−
SC205	28	-5.220205E+01	6.6E-10	19.95	15.14	−
SCAGR7	24	-2.331389E+06	4.6E-14	11.28	7.32	−
SHARE2B	26	-4.157319E+02	1.7E-12	14.73	7.80	−
RECIPE	25	-2.666160E+02	6.9E-09	14.89	2.20	−−
VTP.BASE	25	1.298312E+05	9.6E-08	30.63	6.72	−−
SHARE1B	36	-7.658930E+04	2.0E-13	30.57	25.28	≈
BORE3D	37	1.373081E+03	4.6E-10	69.17	23.82	−−
SCORPION	33	1.878126E+03	3.7E-09	53.77	19.87	−−
CAPRI	35	2.690014E+03	1.2E-14	106.02	32.19	−−
SCAGR25	27	-1.475343E+07	5.9E-14	44.73	91.79	++
SCTAP1	34	1.412251E+03	5.6E-13	49.75	37.33	−
BRANDY	31	1.518511E+03	4.7E-08	73.79	78.95	≈
ISRAEL	36	-8.966445E+05	3.9E-11	102.26	38.20	−−
ETAMACRO	42	-7.557145E+02	1.2E-09	327.15	106.96	−−
SCFXM1	35	1.841677E+04	1.9E-08	94.31	72.68	−
GROW7	27	-4.778780E+07	1.2E-15	49.19	42.67	≈
BANDM	38	-1.586280E+02	1.3E-10	103.73	107.71	≈
E226	41	-1.875191E+01	1.1E-09	91.65	72.75	−
STANDATA	44	1.257701E+03	7.3E-10	107.47	17.47	−−
SCSD1	24	8.666743E+00	2.7E-12	33.58	38.28	≈
GFRD-PNC	26	6.902242E+06	1.0E-09	57.51	206.55	++
BEACONFD	25	3.359250E+04	4.7E-10	62.74	14.10	−−
STAIR	32	-2.512668E+02	7.7E-12	338.5	190.08	−
SCRS8	48	9.043039E+02	6.1E-10	185.49	177.86	≈
SEBA	32	1.571150E+04	5.1E-06	111.26	106.56	≈
SHELL	34	1.208845E+09	5.1E-07	114.39	78.57	−
PILOT4	61	-2.581134E+03	1.5E-09	736.22	656.83	≈
SCFXM2	39	3.666027E+04	5.1E-09	211.76	319.19	+
SCSD6	25	5.050012E+01	1.1E-12	61.62	164.71	++
GROW15	29	-1.068709E+08	9.5E-14	116.58	194.65	+
SHIP04S	48	1.798716E+06	2.0E-10	152.21	35.20	−−
FFFFF800	56	5.556472E+05	1.1E-10	681.61	281.97	−−
GANGES	24	-1.095857E+05	5.3E-10	503.57	372.73	−
SCFXM3	38	5.490129E+04	8.7E-09	313.14	632.04	++
SCTAP2	35	1.724809E+03	7.2E-12	352.05	342.76	≈
GROW22	33	-1.608343E+08	5.7E-14	192.68	403.74	++
SHIP04L	37	1.793326E+06	5.3E-09	170.33	67.03	−−
PILOT.WE	65	-2.720078E+06	7.7E-10	814.49	3850.05	++
SIERRA	34	1.539483E+07	7.4E-11	280.41	700.02	++
SHIP08S	62	1.920099E+06	1.3E-09	335.36	113.50	−−
SCTAP3	36	1.424001E+03	1.9E-11	422.54	570.60	+
SHIP12S	39	1.489237E+06	2.4E-09	272.62	274.72	≈
25FV47	47	5.501849E+03	1.7E-14	1338.47	5722.41	++
SCSD8	22	9.050008E+02	1.0E-13	112.67	1174.23	++
NESM	43	1.407605E+07	1.3E-13	744.05	1296.87	+
CZPROB	59	2.185198E+06	7.9E-10	431.12	836.44	+
PILOTJA	82	-6.112604E+03	7.5E-08	5870.06	5496.13	≈
SHIP08L	46	1.909057E+06	5.5E-08	440.48	244.25	−
SHIP12L	41	1.470189E+06	2.5E-09	508.22	621.37	≈
80BAU3B	63	9.872249E+05	7.4E-10	2486.11	11768.52	++
PILOTS	57	-5.574894E+02	7.4E-10	32010.14	74443.58	++

Primal Barrier (Unscaled)

for **MINOS** and a comparison category. Based on the ratio φ = barrier time/MINOS time, the categories stand for

$$
\begin{aligned}
++ &\quad \text{for} \quad \varphi < 0.5 \\
+ &\quad \text{for} \quad 0.5 \leq \varphi < 0.8 \\
\approx &\quad \text{for} \quad 0.8 \leq \varphi < 1.25 \\
- &\quad \text{for} \quad 1.25 \leq \varphi < 2.0 \\
-- &\quad \text{for} \quad \varphi \geq 2.0 \, .
\end{aligned}
$$

Some observations may be made that are generally true for all barrier method runs. The iteration count is low, rarely over 60, and it increases little with the size of the problem. The barrier times are relatively better for larger problems and are especially good for problems that are hard to solve for the simplex code.

Failures

As with any other algorithm, if one set of parameters must be chosen for all problems, the performance is not as good as when the parameters are chosen for a smaller subset of the problems. The difficulty of choosing an acceptable set of parameters is even greater for the primal barrier implementation, where both performance and reliability prove to be a problem. The barrier code using a given set of parameters might fail to solve an LP for a number of reasons:

- *Slow convergence.* If the starting point or any other iterate is not sufficiently interior, the method is likely to take many small steps along the boundary of the feasible region. We terminate the algorithm at iteration 120 and rate such behavior as a failure.

- *No Phase II.* When the objective weight ω is too large, the convergence criteria might be satisfied before a sufficiently feasible point is found.

- *Infeasible termination.* Ill-conditioning in Phase II may result in infeasible search directions, leading to a solution that lies outside of the feasibility tolerance.

- *Overflow.* Extreme ill-conditioning (and/or insufficient remedies for it) may lead to floating-point numbers larger than the maximum machine-representable number during the factorization (1.7×10^{38} in the D_floating format).

	Itn.	Obj. fct.	$\|Ax-b\|/\|x\|$	CPU sec.	MINOS 5.3	
AFIRO	17	-4.647526E+02	3.00E-12	2.29	0.49	−−
ADLITTLE	24	2.255013E+05	4.10E-08	8.66	5.07	−
SC205	23	-5.220215E+01	9.40E-08	14.23	15.14	≈
SCAGR7	26	-2.331375E+06	6.30E-15	11.45	7.32	−
SHARE2B	26	-4.157318E+02	4.40E-12	14.92	7.80	−
RECIPE	17	-2.666160E+02	9.10E-08	9.98	2.20	−−
VTP.BASE	23	1.298310E+05	2.60E-07	28.28	6.72	−−
SHARE1B	34	-7.658889E+04	1.00E-12	27.07	25.28	≈
BORE3D	26	1.373083E+03	6.60E-08	48.75	23.82	−−
SCORPION	21	1.878126E+03	6.10E-08	34.85	19.87	−
CAPRI	29	2.690044E+03	1.80E-10	85.81	32.19	−−
SCAGR25	28	-1.475334E+07	3.70E-14	42.24	91.79	++
SCTAP1	27	1.412254E+03	1.50E-12	37.43	37.33	≈
BRANDY	26	1.518536E+03	8.20E-08	62.28	78.95	+
ISRAEL	34	-8.966053E+05	9.80E-13	64.37	38.20	−
ETAMACRO	30	-7.557044E+02	1.00E-07	236.56	106.96	−−
SCFXM1	30	1.841676E+04	8.50E-08	77.14	72.68	≈
BANDM	29	-1.586245E+02	8.20E-08	75.39	107.71	+
E226	30	-1.875164E+01	6.70E-08	66.69	72.75	≈
STANDATA	34	1.258308E+03	1.10E-07	85.47	17.47	−−
SCSD1	21	8.666740E+00	2.80E-12	27.08	38.28	+
GFRD-PNC	22	6.902595E+06	1.10E-07	47.81	206.55	++
BEACONFD	21	3.359320E+04	7.70E-08	53.38	14.10	−−
STAIR	28	-2.512595E+02	5.70E-11	290.56	190.08	−
SCRS8	34	9.043523E+02	4.40E-08	125.2	177.86	+
SEBA	23	1.571166E+04	2.40E-07	76.43	106.56	+
SHELL	38	1.208829E+09	5.50E-14	113.49	78.57	−
PILOT4	40	-2.58 1 '9E+03	7.10E-08	498.36	656.83	+
SCFXM2	37	3.665 ›)7E+04	6.90E-08	186.09	319.19	+
SCSD6	21	5.050031E+01	3.60E-12	47.74	164.71	++
SHIP04S	27	1.798717E+06	1.20E-08	90.38	35.20	−−
GANGES	24	-1.095840E+05	1.20E-07	510.09	372.73	−
SCFXM3	35	5.490126E+04	6.90E-08	263.77	632.04	++
SCTAP2	26	1.724819E+03	5.10E-15	251.47	342.76	+
SHIP04L	26	1.793327E+06	5.20E-08	124.65	67.03	−
PILOT.WE	43	-2.720058E+06	1.10E-07	539.88	3850.05	++
SIERRA	59	1.541763E+07	1.80E-12	446.75	700.02	+
SHIP08S	26	1.920100E+06	2.40E-08	146.23	113.50	−
SCTAP3	27	1.424016E+03	8.20E-15	301.94	570.60	+
SHIP12S	26	1.489237E+06	4.80E-08	180.33	274.72	+
25FV47	40	5.501945E+03	6.40E-15	1138.12	5722.41	++
SCSD8	20	9.050034E+02	4.00E-11	94.48	1174.23	++
NESM	37	1.407627E+07	3.50E-14	628.35	1296.87	++
CZPROB	49	2.185220E+06	1.10E-07	366.56	836.44	++
SHIP08L	27	1.909057E+06	8.90E-08	259.32	244.25	≈
SHIP12L	27	1.470189E+06	8.80E-08	336.27	621.37	+
80BAU3B	50	9.871698E+05	8.40E-08	1922.1	11768.52	++
PILOTS	56	-5.574775E+02	1.00E-07	31453.99	74443.58	++

Primal Barrier (Scaled)

Our tests of the primal barrier implementation with the 53 *netlib* problems yielded few runs without failures. The parameters of the successful runs were all from a very small neighborhood of the parameter set used for the run on page 65.

Scaling

One strategy that generally improves the algorithmic performance is the use of scaling. This improvement is partly due to the observed fact that scaling usually increases the range of reliable parameters.

On scaled problems we usually observe the resulting $\|x\|$ to be in the order of one. However, the scaling routine used was not successful for three of our test problems (GROW7, GROW15 and GROW22), where $\|x\|$ remained at 10^7. The barrier code subsequently failed because of slow convergence for these problems.

The table on page 67 shows results for a run of scaled problems. In addition to the three problems above, two problems with rank-deficient constraint matrices are not included, namely PILOTJA and FFFFF800.[7] With this reduced test set, the slack variables of equality constraints can be left fixed and the weight in the objective function is chosen to $\omega = 0.1$.

Almost all problems of this set were solved faster with these settings, some considerably so. Several midsize problems show better solution times than those achieved with MINOS, while the simplex code holds its advantage for small problems. Notice, that the MINOS results are also obtained for the scaled problems.

Dense Columns

Only four test problems have dense columns in Phase II according to our definition.

	dense columns	nonzeros
ISRAEL	15	≥ 35
SEBA	14	≥ 185
FFFFF800	1	50
PILOTS	23	≥ 72

[7] The problem FFFFF800 is also omitted in [ARV86], although the authors claim to solve every *netlib*-problem with simple bounds except 25FV47.

More Parameters

In order to analyze the impact of some of the parameters more closely, we shall compare several runs where one parameter is varied while the others stay fixed at some default values. The run for these default parameter values provides the basis of the comparisons. The running times for each problem are categorized as $++/--$ for at least 20% better/worse and $+/-$ for at least 5% better/worse and the total for each category is given. The default values were chosen for their general reliability; they do not necessarily represent the best choice in terms of performance. The test set includes the problems used in the run of page 67, except for PILOTS.[8] Scaling was used in all cases.

Maximal step As explained on page 13, an iterate close to the boundary is avoided by using some maximal step $\phi_\alpha \alpha_M$ instead of the theoretical maximum α_M. The usual value for this factor is $\phi_\alpha = 0.98$.

$\phi_\alpha =$	0.88	0.95	0.97	[0.98]	0.99
$++$	2	1	0	0	0
$+$	0	3	4	0	8
$\approx$	13	28	33	47	36
$-$	30	14	9	0	3
$--$	2	1	1	0	0

Unless the steplength is limited severely, the impact of this factor is marginal. The absence of failures in the column for $\phi_\alpha = 0.99$ indicates that the linesearch procedure is working well.

Free variables. The penalty parameter ρ of the approximated least-squares problem implies bounds for free variables at a distance of $\sqrt{2\mu\rho}$ (see page 18).

$\rho =$	10^3	10^5	10^7	[10^9]	10^{11}
$++$	2	2	1	0	0
$+$	1	20	12	0	1
$\approx$	7	17	33	47	28
$-$	18	5	0	0	17
$--$	16	0	0	0	1
failed	3	1	1	0	1

Small values of ρ impede convergence by limiting the rate of change in free variables, while large values may generate a large fluctuation in their values. Consequently a good choice for ρ is higher for unscaled problems than for scaled problems.

[8] PILOTS was not included solely because of its run time, 9 hours, which would have unnecessarily decreased the number of possible test runs. Otherwise, no special difficulty was encountered when running PILOTS.

Composite objective function. The weight ω of ALP on page 19 has an almost monotonic effect on the performance.

$\omega =$	1.0	[0.1]	0.01	0.001
++	0	0	0	0
+	21	0	3	2
$\approx$	20	47	31	18
−	0	0	12	25
−−	2	0	1	0
failed	4	0	0	2

In most cases it is advantageous to increase ω up to a neighborhood of the problem dependent bound ω', after which the algorithm fails to find a solution.

Starting point. The size ν of the linear modification of the barrier term, page 24, is most important for the choice of the starting point at a distance of $1/\nu$ to the bounds.

$\nu =$	1.0	0.1	[0.01]	0.001	0.0001
++	10	0	0	1	0
+	27	25	0	1	2
$\approx$	6	19	47	17	3
−	2	2	0	27	31
−−	1	1	0	1	3
failed	1	0	0	0	6

The impact of ν is highly dependent on the choice of μ^1 and ω. The fact that the algorithm performs well with large values of ν is mostly due to the effect of scaling. If the resulting starting point is not sufficiently interior, the number of iterations may be substantial.

Barrier Parameter. The μ of the first subproblem is μ^1 multiplied by $c^T x/n$.

$\mu^1 =$	1.0	0.1	[0.01]	0.001	0.0001
++	4	1	0	0	0
+	0	3	0	4	8
$\approx$	19	35	47	41	36
−	20	5	0	2	3
−−	3	0	0	0	0
failed	1	3	0	0	0

The effect of different choices for μ^1 is minimal on a fairly large interval. The boundaries of this interval depend heavily on ν.

Bibliography

[AKRV87] I. Adler, N. Karmarkar, M. G. C. Resende and G. Veiga (1987). Data structures and programming techniques for the implementation of Karmarkar's algorithm, Manuscript (December 1987), Department of Industrial Engineering and Operations Research, University of California, Berkeley, CA.

[ARV86] I. Adler, M. G. C. Resende and G. Veiga (1986). An implementation of Karmarkar's algorithm for linear programming, Report ORC 86-8, Department of Industrial Engineering and Operations Research, University of California, Berkeley, CA.

[Bj87a] Å. Björck (1987). Stability analysis of the method of semi-normal-equations for linear least-squares problems, *Linear Algebra and its Applications*, 88/89, 31–48.

[Bj87b] Å. Björck (1987). Iterative refinement and reliable computing, *Advances in Reliable Numerical Computing*, M. Cox and S. J. Hammarling, ed., Oxford University Press.

[Brent73] R. P. Brent (1973). *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, NJ.

[CGLN84] E. Chu, J. A. George, J. W. H. Liu and E. Ng (1984). SPARSPAK: Waterloo Sparse Matrix Package User's Guide for SPARSPAK-A, Report CS-84-36, Department of Computer Science, University of Waterloo, Waterloo, Canada.

[Dan63] G. B. Dantzig (1963). *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey.

[DER86] I. S. Duff, A. M. Erisman and J. K. Reid (1986). *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford, England.

[Dik67] I. I. Dikin (1967). Iterative solution of problems of linear and quadratic programming, *Doklady Akademii Nauk SSSR*, Tom 174, No. 4.

[Fia79] A. V. Fiacco (1979). Barrier methods for nonlinear programming, in A. Holzman (ed.), *Operations Research Support Methodology*, Marcel Dekker, Inc., New York, 377–440.

[FM68] A. V. Fiacco and G. P. McCormick (1968). *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*, John Wiley and Sons, New York and Toronto.

[Fle81] R. Fletcher (1981). *Practical Methods of Optimization, Volume 2*, John Wiley and Sons, Chichester and New York.

[FM69] R. Fletcher and A. P. McCann (1969). Acceleration techniques for nonlinear programming, in: R. Fletcher, ed., *Optimization*, Academic Press, London, 203–213.

[Fri54] K. R. Frisch (1954). Principles of linear programming—with particular reference to the double gradient form of the logarithmic barrier function, Memorandum, University Institute of Economics, Oslo, Norway.

[Fri57] K. R. Frisch (1957). Linear dependencies and a mechanized form of the multiplex method for linear programming, University Institute of Economics, Oslo, Norway.

[Gay85] D. M. Gay (1985). Electronic mail distribution of linear programming test problems, *Mathematical Programming Society COAL Newsletter* 13, 10–12.

[Gay88] D. M. Gay (1988). Massive memory buys little speed for complete, in-core sparse Cholesky factorizations, Numerical Analysis Manuscript 88-04, AT&T Bell Laboratories, Murray Hill, NJ 07974.

[GL81] J. A. George and J. W. H. Liu (1981). *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, NJ.

[GL87] J. A. George and J. W. H. Liu (1987). The evolution of the minimum degree ordering algorithm, ORNL/TM-10452, Oak Ridge National Laboratory, Oak Ridge, TN.

[GLN88] J. A. George, J. W. H. Liu and E. Ng (1988). A data structure for sparse QR and LU factorizations, *SIAM J. on Scientific and Statistical Computing* 9, 100–121.

[GMSTW86] P. E. Gill, W. Murray, M. A. Saunders, J. A. Tomlin and M. H. Wright (1986). On projected Newton barrier methods for linear programming and an equivalence to Karmarkar's projective method, *Mathematical Programming* 36, 183–209.

[GMSW84] P. E. Gill, W. Murray, M. A. Saunders and M. H. Wright (1984). Sparse matrix methods in optimization, *SIAM J. on Scientific and Statistical Computing* 5, 562–589.

[GMSW86] P. E. Gill, W. Murray, M. A. Saunders and M. H. Wright (1986). A note on nonlinear approaches to linear programming, Report SOL 86-7, Department of Operations Research, Stanford University, Stanford, CA.

[GMSW88] P. E. Gill, W. Murray, M. A. Saunders and M. H. Wright (1988). A practical anti-cycling procedure for linear and nonlinear programming, Report SOL 88-4, Department of Operations Research, Stanford University, Stanford, CA.

[GMW81] P. E. Gill, W. Murray and M. H. Wright (1981). *Practical Optimization*, Academic Press, London and New York.

[GN84] J. A. George and E. Ng (1984). SPARSPAK: Waterloo sparse matrix package user's guide for SPARSPAK-B, Report CS-84-37, Department of Computer Science, University of Waterloo, Waterloo, Canada.

[Gon87] C. C. Gonzaga (1987). Search directions for interior linear programming methods, Memorandum UCB/ERL M87/44, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA.

[GVL83] G. H. Golub and C. F. Van Loan (1983). *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MA.

[HS52] M. R. Hestenes and E. Stiefel (1952). Methods of conjugate gradients for solving linear systems, *J. Res. Nat. Bur. Stand.* 49, 409–436.

[Jit78] K. Jittorntrum (1978). *Sequential Algorithms in Nonlinear Programming*, Ph. D. Thesis, Australian National University, Canberra, Australia.

[JO78] K. Jittorntrum and M. R. Osborne (1978). Trajectory analysis and extrapolation in barrier function methods, *Journal of Australian Mathematical Society* 20 (Series B) 352–369.

[Kar84] N. K. Karmarkar (1984). A new polynomial-time algorithm for linear programming, *Combinatorica* 4, 373–395.

[KR88] N. K. Karmarkar and K. G. Ramakrishnan (1988). Implementation and computational results of the Karmarkar algorithm for linear programming, using an iterative method for computing projections, extended abstract for the Mathematical Programming Symposium, Tokyo, Japan.

[Kha79] L. G. Khachiyan (1979). A polynomial algorithm in linear programming, *Doklady Akademii Nauk SSSR* Novaia Seriia 244, 1093–1096. [English translation in *Soviet Mathematics Doklady* 20, 191–194.]

[Lus87] I. J. Lustig (1987). An analysis of an available set of linear programming test problems, Report SOL 87-11, Department of Operations Research, Stanford University, Stanford, CA.

[Mon87] C. L. Monma (1987). Recent breakthroughs in linear programming methods, Manuscript, Bell Communications Research, Morristown, NJ.

[MM87] C. L. Monma and A. J. Morton (1987). Computational experience with a dual affine variant of Karmarkar's method for linear programming, *Operations Research Letters* 6, 261–267.

[MA87] R. D. C. Monteiro and I. Adler (1987). Interior path following primal-dual algorithms—Part I: Linear programming, Technical Report, Department of Industrial Engineering and Operations Research, University of California, Berkeley, CA.

[MS83] B. A. Murtagh and M. A. Saunders (1983). MINOS 5.0 user's guide, Report SOL 83-20, Department of Operations Research, Stanford University, Stanford, CA.

[MW76] W. Murray and M. H. Wright (1976). Efficient linear search algorithms for the logarithmic barrier function, Report SOL 76-18, Department of Operations Research, Stanford University, Stanford, CA.

[Osb72] M. R. Osborne (1972). Topics in Optimization, Report CS-72-279, Computer Science Department, Stanford University, Stanford, CA.

[PS82] C. C. Paige and M. A. Saunders (1982). LSQR: An algorithm for sparse linear equations and sparse least squares, *ACM Transactions on Mathematical Software* 8, 43–71.

[Par61] G. R. Parisot (1961). *Résolution Numérique Approchée du Problème du Programmation Lineaire par Application de la Programmation Logarithmique*, Ph. D. Thesis, University of Lille, France.

[RS88] J. Renegar and M. Shub (1988). Simplified complexity analysis for Newton LP methods, Technical Report No. 807, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY.

[Shor77] N. Z. Shor (1977). The cut-off method with space dilation for solving convex programming problems, *Kibernetika* 13, No. 1, 94–95. [English translation in *Cybernetics* 13 (1978), 94–96.]

[Stew87] G. W. Stewart (1987). An iterative method for solving linear inequalities. Report TR-1833, Department of Computer Science, University of Maryland, College Park, MD.

[Van89] R. J. Vanderbei (1989). Affine-scaling for linear programs with free variables, *Mathematical Programming* 43, 31–44.

[VMF86] R. J. Vanderbei, M. S. Meketon and B. A. Freedman (1986). A modification of Karmarkar's linear programming algorithm, *Algorithmica* 1, 395–407.

[VLo85] C. F. Van Loan (1985). On the method of weighting for equality-constrained least-squares problems, *SIAM Journal on Numerical Analysis* 22, 851–864.

[Wri76] M. H. Wright (1976). Numerical methods for nonlinearly constrained optimization, Report CS-76-566, Computer Science Department, Stanford University, Stanford, CA.

[Yan81] M. Yannakakis (1981). Computing the minimum fill-in is NP-complete, *SIAM J. Alg. & Disc. Math.*, vol. 2, pp. 77–79.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER SOL 89-6	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Primal Barrier Methods for Linear Programming		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Aeneas Marxen		8. CONTRACT OR GRANT NUMBER(s) N00014-87-K-0142
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Operations Research - SOL Stanford University Stanford, CA 94305-4022		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 1111MA
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research - Dept. of the Navy 800 N. Quincy Street Arlington, VA 22217		12. REPORT DATE June 1989 13. NUMBER OF PAGES 75 pages
		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Linear Programming, Barrier Functions, Cholesky's Algorithm, Sparse Matrices.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(see reverse side)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Primal Barrier Methods for Linear Programming
Aeneas Marxen - SOL 89-6 ABSTRACT

The linear program $\min c^T x$ subject to $Ax = b$, $x \geq 0$, is solved by the *projected Newton barrier method*. The method consists of solving a sequence of subproblems of the form $\min c^T x - \mu \sum \ln x_j$ subject to $Ax = b$. Extensions for upper bounds, free and fixed variables are given. A linear modification is made to the logarithmic barrier function, which results in the solution being bounded in all cases. It also facilitates the provision of a good starting point. The solution of each subproblem involves repeatedly computing a search direction and taking a step along this direction. Ways to find an initial feasible solution, step sizes and convergence criteria are discussed.

Like other *interior-point methods* for linear programming, this method solves a system of the form $AH^{-1}A^T q = y$, where H is diagonal. This system can be very ill-conditioned and special precautions must be taken for the Cholesky factorization. The matrix A is assumed to be large and sparse. Data structures and algorithms for the sparse factorization are explained. In particular, the consequences of relatively dense columns in A are investigated and a Schur-complement method is introduced to maintain the speed of the method in these cases.

An implementation of the method was developed as part of the research. Results of extensive testing with medium to large problems are presented and the testing methodologies used are discussed.